Università degli Studi di Udine
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Corso di Laurea in Informatica

Tesi di Laurea

Un simulatore di k-MdT in Java a fini didattici

Candidato
Burigana Alessandro

Relatore
Prof. Dovier Agostino

Anno accademico 2016-2017

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
http://www.dimi.uniud.it/

Sommario

Con questo testo si intende descrivere il simulatore di k-MdT *A-machine*, realizzato a fini didattici a sostegno del corso di Fondamenti dell'Informatica dell'Università degli Studi di Udine e di qualsiasi altro corso faccia uso del formalismo della macchina di Turing. A tal fine viene fornita una panoramica del contesto storico che ha portato Alan Turing, nel 1936, a scrivere il suo più famoso articolo *On computable numbers, with an application to the Entscheidungsproblem*, a partire dalla crisi dei fondamenti della matematica e procedendo con una breve digressione storica sulle opere di Leibniz, Boole e Frege.

Nel capitolo 2 verrà descritto più nel dettaglio il formalismo di Turing, per poi passare ad una presentazione delle sintassi più utilizzate per le macchine di Turing. Nel capitolo 3 verrà fornita una panoramica su simulatori esistenti, per giungere infine alla effettiva descrizione del programma oggetto di questa dissertazione, alla quale saranno accompagnate delle immagini descrittive e dei programmi di esempio, che verranno forniti insieme all'eseguibile al fine di creare una ridotta ma sostanziale libreria.

Indice

Indice v

Elenco delle figure vii

1 Il contesto storico 1

1.1 La crisi dei fondamenti della matematica . . . 2

1.2 L'*Entscheidungsproblem* . . . 3

2 La Macchina di Turing 7

2.1 On Computable Numbers . . . 7

2.2 Una definizione formale . . . 11

2.3 Simulatori esistenti . . . 13

3 A-Machine 15

3.1 Macchine esterne . . . 17

3.2 Altre funzioni . . . 19

Bibliografia 21

Elenco delle figure

3.1 Interfaccia grafica di `A-Machine` 15
3.2 Una macchina di I/O 17

1

Il contesto storico

> Il pericolo che l'intera costruzione teorica potesse dissolversi nell'inesattezza era [...] bene o male scongiurato nella concezione classica dall'evidenza intuitiva degli elementi primitivi. Venuta a mancare questa, o almeno la fiducia in essa, ecco che quel pericolo si affacciava con tutta la sua forza.
>
> *E. Casari*

8 agosto 1900, Sorbonne, Università di Parigi. David Hilbert, matematico tedesco dell'università di Göttingen, espone al secondo Congresso internazionale dei matematici una lista di dieci[1] problemi aperti, molti dei quali avrebbero influenzato la matematica del XX secolo. Tra questi i primi due erano particolarmente legati alla questione dei fondamenti della matematica: il primo riguardava l'ipotesi del continuo, formulata da Georg Cantor[2], che afferma che *non esiste nessun insieme la cui cardinalità è strettamente compresa fra quella dei numeri interi* ($\aleph_0$) *e quella dei numeri reali* (C); il secondo chiedeva, invece, di provare che gli assiomi dell'aritmetica fossero consistenti, ovvero di dimostrare che partendo da tali assiomi non fosse possibile dedurre due proposizioni che fossero l'una la negazione logica dell'altra.

Non c'è da stupirsi del fatto che Hilbert abbia aperto la sua lista con due problemi mirati alla sistemazione delle nuove discipline: la matematica dell'ultimo trentennio dell'Ottocento aveva subito trasformazioni radicali, come lo spostamento dei fondamenti della matematica dall'intuizione alla logica. Si voleva, dunque, che la matematica assumesse un *rigore* che ancora le mancava, abbandonando l'intuizione come fonte di verità.

All'epoca, infatti, i principi della matematica classica erano intesi come delle *verità rivelate*, lungi da qualsiasi necessità di giustificazione. Come se non bastasse, queste *verità* venivano adoperate per accettare o confutare altre tesi. Hilbert si accorse ben presto di questo eccesso di libertà e con lui diversi altri matematici del tempo. Uno fra tutti fu Bertrand Russell, il quale, leggendo i *Principi* del matematico Gottlob Frege, notò una contraddizione all'interno del sistema di assiomi adoperato proprio in quello scritto. Il 16 giugno 1902 Russell scrisse una lettera a Frege in cui gli comunicava, insieme alla sua ammirazione per le sue opere, di aver *trovato una difficoltà* in un punto: il V assioma dei *Principi*,

[1] La lista completa contiene ventitré problemi, ma per ragioni di tempo Hilbert dovette selezionarne solo alcuni.

[2] Georg Cantor, matematico tedesco e padre della teoria degli insiemi, era convinto della validità dell'ipotesi del continuo e per diversi anni tentò, senza però mai riuscirci, di dimostrarla.

infatti, permetteva di arrivare ad una contraddizione, che diventerà nota come l'antinomia di Russell e che minava i fondamenti della teoria degli insiemi. Essa afferma:

> «L'insieme di tutti gli insiemi che non appartengono a sé stessi appartiene a sé stesso se e solo se non appartiene a sé stesso.»

Infatti, sia T l'insieme appena descritto, ovvero tale da contenere tutti gli insiemi che non sono elementi di loro stessi. Se T appartiene a sé stesso ($T \in T$), allora per definizione T è un insieme che non appartiene a sé stesso. Ma allora, poiché $T \notin T$, ancora per definizione si ha che T appartiene a sé stesso.

Per di più, questa contraddizione scovata da Russell era accompagnata da diverse altre e ciò rese evidente la necessità di riformulare in maniera rigorosa e formale la teoria insiemistica e, con essa, l'intera matematica. Per questo motivo, alla lettera di Russell si è soliti far ricondurre l'inizio della *crisi dei fondamenti della matematica.*

1.1 La crisi dei fondamenti della matematica

Dopo che il contenuto della lettera che Russel inviò a Frege divenne pubblico, infatti, si iniziò a vedere la difficoltà di fondare la matematica come una vera e propria *crisi* dei fondamenti. È solo negli anni Venti, però, che Hilbert decise, insieme ai suoi allievi, di affrontare effettivamente il problema. Nacque così il Programma di Hilbert, che mirava alla formalizzazione di tutte le teorie matematiche esistenti attraverso un insieme finito di assiomi, e alla dimostrazione che questi assiomi non conducevano a contraddizioni. Hilbert propose che la non contradditorietà di sistemi complessi potesse essere provata in termini di sistemi più semplici, fino a basare l'intera matematica sull'aritmetica. Fino a quel tempo, infatti, le prove di non contradditorietà non erano altro che prove di coerenza relativa, ovvero riducevano la coerenza di un insieme di assiomi a un altro, ma Hilbert voleva giungere al capolinea.

Il fine ultimo del Programma di Hilbert, dunque, era quello di creare dei saldi fondamenti della matematica, estirpare l'intuizione come strumento matematico e lasciare spazio unicamente alla logica e al rigore. Per giungere a tale scopo Hilbert incluse vari punti nel suo programma:

- *La formalizzazione di tutta la matematica*, che prevedeva la riscrittura di tutte le formule matematiche in un preciso linguaggio formale e la manipolazione di queste secondo regole ben definite.

- *Completezza*: la prova che tutte le formule matematiche vere possano essere provate all'interno del formalismo stesso.

- *Non contradditorietà*: la prova che non si possa riscontrare alcuna contraddizione all'interno del formalismo della matematica.

- *Conservazione*: la prova che ogni risultato riguardante *oggetti reali* ottenuto ragionando su *oggetti ideali* (come insiemi non numerabili) possa essere provato senza usare oggetti ideali.

- *Decidibilità*: la formulazione di un algoritmo in grado di decidere la verità o la falsità di ogni formula matematica.

Quest'ultimo punto del programma di Hilbert prende nome di *Entscheidungsproblem* e verrà approfondito nella prossima sezione.

Il Programma hilbertiano, tuttavia, incontrò ben presto delle difficoltà, poiché già nel 1931 il logico Kurt Gödel aveva formulato i suoi due teoremi di incompletezza [15]. Il primo afferma che in ogni formalizzazione coerente della matematica che sia sufficientemente potente da poter assiomatizzare la teoria elementare dei numeri naturali – vale a dire, sufficientemente potente da definire la struttura dei numeri naturali dotati delle operazioni di somma e prodotto – è possibile costruire una proposizione sintatticamente corretta che non può essere né dimostrata né confutata all'interno dello stesso sistema. Il secondo teorema di incompletezza di Gödel, invece, afferma che nessun sistema, che sia abbastanza espressivo da contenere l'aritmetica, può essere utilizzato per dimostrare la sua stessa coerenza.

In sostanza, il primo teorema di incompletezza di Gödel dimostra che qualsiasi sistema che permette di definire i numeri naturali e le relative operazioni di somma e prodotto è necessariamente incompleto. In altre parole, esso contiene affermazioni di cui non si può dimostrare né la verità, né la falsità. Inoltre, Gödel mostra come non si possa mai ottenere la definizione di una lista completa di assiomi che consenta di dimostrare tutte le verità.

Il secondo teorema di incompletezza di Gödel mostra che, poiché nemmeno un sistema particolarmente semplice come quello dell'aritmetica elementare può essere utilizzato per provare la propria coerenza, certamente non potrà essere utilizzato per dimostrare la coerenza di sistemi più potenti.

Come se non bastasse, mettere ulteriormente i bastoni tra le ruote al programma di Hilbert fu Alan Turing, che nel 1936 scrisse un articolo in cui dimostrava che nemmeno l'*Entscheidungsproblem* aveva una soluzione.

1.2 L'*Entscheidungsproblem*

Nel 1928, al Congresso internazionale dei matematici di Bologna, David Hilbert espose tre quesiti, il terzo dei quali è conosciuto con il nome di *Entscheidungsproblem*, o *Problema della decisione* (*Decision problem*). Tale quesito richiede di fornire un algoritmo che prenda in input una formula della logica del primo ordine e che ritorni *Sì*, se la formula è universalmente valida e *No* altrimenti.

Ma le radici di questo problema risalgono a tempi ben più lontani. Fu infatti il matematico e filosofo Gottfried Liebniz, nato a Lipsia nel 1646, a gettarne le fondamenta. Nel 1673, a ventisette anni, Liebniz presentava un modello di macchina calcolatrice che superava le capacità della famosa Pascalina (1642) dell'omonimo matematico. Infatti, oltre alle operazioni di addizione e sottrazione, il congegno di Liebniz era in grado di eseguire moltiplicazioni e divisioni. Tale dispositivo comprendeva un meccanismo noto come "ruota di Liebniz", sfruttato anche in diverse macchine meccaniche del XX secolo.

Nonostante la creazione del matematico di Lipsia consentisse solo di svolgere operazioni di aritmetica ordinaria, il fatto stesso di poter meccanizzare il calcolo aveva un significato ben più ampio. Già nel 1674, infatti, Liebniz descriveva una macchina in grado di risolvere equazioni algebriche e, un anno più tardi, paragonava il ragionamento logico ad un mero meccanismo. Ma per automatizzare il pensiero umano era necessario un linguaggio ben strutturato, un *alfabeto delle idee*. Liebniz chiamava un tale insieme di simboli *reale*, in grado di abbracciare il pensiero umano in tutta la sua estensione, una *caratteristica universale*.

L'idea di Liebniz, più nel dettaglio, consisteva in un insieme di simboli accuratamente selezionati[3] che riflettessero concetti reali e a cui affiancare delle operazioni logiche, il *calculus ratiocinator*, in grado di manipolare tali simboli. Scrive Louis Couturat, logico e studioso di Leibniz:

> «È la notazione algebrica a incarnare, per così dire, l'ideale della caratteristica e a servire da modello; ed è ancora l'algebra che Leibniz porta ad esempio per mostrare come un sistema di simboli ben scelti sia utile e anzi indispensabile per il pensiero deduttivo.»

Un tale linguaggio consentiva, dunque, di stabilire mediante calcoli simbolici la verità di un qualsiasi enunciato e le relazioni logiche che intercorrevano fra questi. Ecco come, quindi, il problema della decisione vede il suo fautore proprio in Gottfried Liebniz. Scrive Martin Davis ne *Il Calcolatore Universale*:

> «In linea di principio l'algoritmo per l'*Entscheidungsproblem* avrebbe dovuto ridurre tutti i ragionamenti deduttivi umani a calcolo bruto, realizzando in buona misura il sogno di Leibniz.»

Per realizzare quest'enorme opera, il filosofo e matematico di Lipsia decise di suddividerla in tre sottoproblemi: innanzitutto era necessario, al fine di poter scegliere i simboli più adatti, creare un'enciclopedia che comprendesse *tutta* la conoscenza umana; dopodichè sarebbe stata possibile la scelta delle nozioni fondamentali e dei simboli che meglio le rappresentavano; infine, si sarebbero potute ridurre le regole deduttive a manipolazioni di questi simboli. Tuttavia, nel corso della sua vita, Leibniz non riuscì nell'impresa di realizzare un tale linguaggio. Si dovettero aspettare quasi due secoli affinchè si ottenessero dei progressi in questo senso, con l'opera di George Boole[4], logico inglese del XIX secolo, che riuscì a dimostrare come la deduzione logica potesse essere considerata come un ramo della matematica. Boole prese in mano la logica del tempo, quella classica di Aristotele[5], che studiava enunciati come *Tutti gli uomini sono mortali*, *Nessun cane ha le ali*, *Alcuni laghi sono salati*, e comprese che parole come "uomini", "cane" e "laghi" rappresentavano la *classe* degli individui od oggetti descritti dalla parola stessa.

Boole rappresentava le classi con delle lettere, come per le variabili nell'algebra classica, e ne descrisse le operazioni fondamentali. Ad esempio, se x indica la classe di tutte le cose bianche e y quella di tutti gli animali, allora la moltiplicazione xy rappresenterà la classe degli animali bianchi (operazione di *intersezione*). Allo stesso modo definì le operazioni di somma e differenza: se x e y sono due classi, $x + y$ rappresenta la classe di tutte le cose presenti in x o in y (operazione di *unione*), mentre $x - y$ indica la classe degli oggetti contenuti in x, ma non in y.

Osservando l'operazione di moltiplicazione, il logico inglese notò una differenza sostanziale rispetto alla moltiplicazione dell'algebra classica: se, ad esempio, x è la classe di tutte le piante, allora xx indica la classe di tutte le piante che sono piante, dunque rappresenta x stessa. Nell'algebra di Boole, dunque,

[3] Il linguaggio teorizzato da Liebniz prevedeva di adottare dei simboli che fossero in grado di descrivere al meglio il concetto a cui si riferiscono. Fu proprio Liebniz ad introdurre la simbologia del calcolo infinitesimale delle operazioni di derivazione, d, e integrazione, $\int$, in uso ancora oggi. Si può notare come i simboli rappresentino a tutti gli effetti i *concetti* di derivata e di integrale. Questa notazione era nettamente superiore a quella adottata da Newton, che rendeva più macchinose operazioni come, ad esempio, quella di sostituzione.

[4] I risultati di Boole trattati in questo testo si rifanno all'opera *The Mathematical Analysis of Logic* [8].

[5] Sebbene fossero stati fatti alcuni progressi, la logica del tempo di Boole era rimasta pressoché immutata per duemila anni dopo la formalizzazione di Aristotele nell'Organon.

l'equazione $xx = x$ era vera per ogni classe x e ciò lo spinse a determinare i valori di x che soddisfacessero tale equazione. Il risultato immediato che ottenne fu $x = 0$ e $x = 1$ e, dunque, concluse che l'algebra della logica non è altro che quella classica limitata ai soli valori 0 e 1. Il logico inglese si rese conto che, a fini di coerenza, era necessario fornire un'interpretazione per i simboli 0 e 1.

Per fare ciò, osservò il comportamento di questi valori rispetto alla moltiplicazione: $0x = 0$ e $1x = x$. Dunque il simbolo 0 indicherà quella classe a cui non appartiene nulla, la *classe vuota*, mentre il simbolo 1 rappresenterà la classe che contiene tutti gli oggetti, la *classe universo*.

Quello che aveva ottenuto Boole era un sistema logico che superava quello di Aristotele, ma che era ancora lontano dal sogno di Leibniz. Piuttosto che la *caratteristica universale*, egli aveva infatti creato il *calculus ratiocinator*, ovvero l'insieme delle operazioni eseguibili sugli enunciati logici. Inoltre, il sistema logico di Boole non era sufficientemente espressivo. Si consideri l'enunciato "Tutti gli studenti bocciati sono stupidi o pigri". È possibile considerarlo della forma *Tutti gli X sono Y*, rendendo la classe degli studenti stupidi o pigri un'unica entità, cosicché non sia possibile distinguere fra studenti stupidi e studenti pigri.

A descrivere un sistema logico più raffinato, comprensivo di questo tipo più sottile di ragionamento fu Gottlob Frege, matematico, logico e filosofo tedesco. L'obiettivo di Frege era quello di eliminare dal ragionamento matematico le approssimazioni e le arbitrarietà che spesso nascono dal ricorso all'intuizione, conferendo al ragionamento matematico quel *rigore* di cui si è parlato sopra. Per raggiungere tale scopo, egli aveva dunque bisogno di uno strumento formale, un linguaggio logico, che però non poteva essere l'algebra di Boole, perchè in essa operazioni diverse sono rappresentate dallo stesso simbolo, lasciando spazio alle ambiguità (si pensi al simbolo $+$, usato sia per la somma aritmetica, sia per l'operazione di unione fra classi).

Frege si richiama a Leibniz che voleva unire a un calcolo (*calculus ratiocinator*) una vera e propria lingua (caratteristica) universale tanto espressiva da permettere di rappresentare ogni ragionamento: rifacendosi all'idea di Leibniz, nella sua più celebre opera, l'*Ideografia* (*Begriffsschrift* [12]) del 1879, Frege descrisse così un vero e proprio linguaggio, che oggi è chiamato logica del primo ordine. Insieme a questo linguaggio, egli descrisse delle *regole di inferenza*, che consentivano di dedurre proposizioni a partire da altre proposizioni. La più importante di queste, a titolo di esempio, è la seguente: presi x e y enunciati qualsiasi dell'Ideografia, se sono asseriti sia x, sia $x \to y$ (dove $\to$ è l'implicazione logica), allora si può asserire anche y.

In sostanza, quello che Frege aveva ottenuto era un linguaggio in grado di abbracciare tutti i ragionamenti matematici[6], ma era ancora distante dall'idea leibniziana. Infatti, non sempre è possibile giungere ad una conclusione a partire da alcune premesse, mediante le regole descritte da Frege: in questi casi non vi è modo di stabilire se ciò è dovuto ad un limite delle capacità di pensiero o di costanza umani o se, effettivamente, non è possibile ottenere quella conclusione. Inoltre, il meccanismo deduttivo descritto da Frege era troppo macchinoso per poter essere paragonato all'efficienza del linguaggio di Leibniz.

Oltre all'Ideografia, Frege compose svariate opere, fra cui *I Fondamenti dell'aritmetica* (*Die Grundlagen der Arithmetik* [13]) e *I Principi dell'aritmetica* (*Grundgesetze der Arithmetik* [14]), in cui si

[6] La completezza delle regole di Frege venne dimostrata nel 1930 da Kurt Gödel.

proponeva di fondare l'aritmetica su basi logiche. Frege apparteneva infatti alla corrente dei logicisti[7], di cui è considerato il primo esponente, insieme ad altri matematici come Russell e Whitehead. La lettera mandata da Russell di cui si è già discusso in precedenza era relativa ai *Principi*, di cui Frege stava curando la seconda edizione. Non appena ne lesse il contenuto, Frege aggiunse alla sua opera un'appendice in cui riportava l'antinomia di Russell e abbozzò una soluzione, ritenuta insoddisfacente dallo stesso Frege e poi abbandonata. Egli si rese conto di aver fallito nel suo intento e decise, così, di abbandonare il paradigma logicista.

Nel frattempo, Russell e Whitehead stavano lavorando ad un'opera colossale, che condivideva gli scopi dei *Principi* di Frege: si trattava dei *Principia Mathematica* (1910) [7], che riprendevano le idee di Frege. Il sistema logico dei *Principia* fu utilizzato come punto di partenza da David Hilbert e Wilhelm Ackermann nel sopraccitato programma di Hilbert. Come è già stato riportato sopra, uno dei punti di questo programma è quello della decidibilità, con il quale Hilbert voleva ottenere un algoritmo che consentisse di automatizzare le dimostrazioni dei teoremi. Tuttavia, all'epoca, la nozione di algoritmo non era stata ancora formalmente definita. Durante gli anni '30, a dare un contributo in questo senso furono soprattutto Alonzo Church, che nell'articolo *An Unsolvable Problem of Elementary Number Theory* elabora il formalismo del λ calcolo e Alan Turing, con la sua omonima macchina, descritta nel lavoro *On Computable Numbers, with an Application to the Entscheidungsproblem.*

[7] Il logicismo di Frege, il formalismo di Hilbert e l'intuizionismo di Brouwer erano le correnti più importanti nel contesto della crisi dei fondamenti della matematica, che si proponevano di realizzare, ognuna a modo suo, una base solida e rigorosa per la matematica.

2

La Macchina di Turing

Alan Mathison Turing nacque a Londra il 23 giugno 1912. Dopo un'infanzia travagliata, cominciò a frequentare la *public school* di Sherborne, a quattordici anni. Nonostante la giovane età le sue inclinazioni matematico-scientifiche erano già ben definite, ma a Sherborne si esaltava più l'importanza dello sport che quella della matematica. All'inizio Turing seguiva distrattamente le lezioni e non si impegnava molto negli studi, ma le cose cambiarono quando strinse amicizia con Christopher Morcom, che invece era uno studente modello e condivideva la passione di Turing per le materie scientifiche. Grazie all'influenza di Morcom, nell'ultimo anno a Sherborne, Alan ebbe ottimi voti e vinse una borsa di studio per il King's College di Cambridge, grazie alla quale aveva diritto a vitto, alloggio e ottanta sterline all'anno.

A Cambridge l'atmosfera era diametralmente opposta rispetto a quella di Sherborne e il talento di Turing potè fiorire senza ostacoli. Qui, infatti, vi lavoravano personaggi come il matematico Godfrey Harold e il fisico matematico e astronomo Sir Arthur Eddington, che nel 1919 guidò una spedizione in Africa occidentale durante la quale fu possibile osservare, grazie ad un'eclisse totale di Sole, la deflessione della luce di una stella causata dall'attrazione gravitazionale solare, fatto che fornì la prima conferma della teoria della relatività generale di Albert Einstein.

Nel 1935, Turing seguiva un corso tenuto dal matematico Max Newman, il quale partecipò al Congresso internazionale dei Matematici di Bologna del 1928 e assistette alla conferenza di Hilbert di cui si è parlato sopra. Pochi anni più tardi, Newman apprese i risultati di Gödel sulle dimostrazioni di incompletezza e, ispirato da questi argomenti, nella primavera del '35 decise di tenere un corso sui fondamenti della matematica che culminava con i risultati di Gödel: fu grazie a queste lezioni che Turing venne a conoscenza dell'*Entscheidungsproblem* e che iniziò a pensare ad un modo per dimostrare che un algoritmo come quello richiesto da Hilbert non esisteva.

2.1 On Computable Numbers

Nel 1936 Turing scrisse il suo celebre articolo *On Computable Numbers, with an Application to the Entscheidungsproblem* [17], con il quale, in poco più di trenta pagine, diede una scossa al mondo matematico del tempo e costituì le basi della teoria della computabilità.

Turing definì i *numeri computabili*[1] come numeri reali la cui espressione in termini di cifre decimali è calcolabile con mezzi finiti o, in altre parole, se i suoi decimali possono essere scritti da una *macchina*. Il modello di macchina qui citato viene spiegato da Turing per mezzo di un'analogia con una persona che svolge un calcolo: le azioni che compirà saranno, ad esempio, leggere dei simboli (parole, numeri) presenti su un foglio, scrivere, cancellare, spostare l'attenzione su altri simboli e così via. Al fine di spogliare questo processo da particolari non rilevanti, Turing immaginò che la persona coinvolta nel calcolo scrivesse su una striscia di carta orizzontale, in cui non fosse possibile scrivere simboli uno sotto l'altro e che fosse suddivisa in celle tali da contenere un unico simbolo. In questo modo, le azioni di lettura e scrittura appena citate si semplificano di molto, lasciando al soggetto le sole opzioni di spostare l'attenzione da una cella a quella adiacente e scrivere o cancellare il simbolo di quella cella.

Tuttavia, come può la persona descritta decidere quale simbolo scrivere in un certo istante? Ciò dipende dal suo *stato mentale*: Turing definisce questi stati *m-configurations*. Dunque una macchina può leggere un simbolo per volta da un nastro (il particolare foglio di carta appena descritto) diviso in celle (*squares*) e, in funzione del simbolo letto in quell'istante e della *m-configuration* in cui si trova, saprà quale azione intraprendere. La coppia composta dal simbolo letto e dallo stato è detta configurazione. Una macchina il cui comportamento ad ogni passo è completamente dettato da una configurazione venne chiamata da Turing una *automatic machine*, o *a-machine*.

Formalmente, il comportamento di una di queste macchine è descritto da delle quintuple del tipo (q_c, S_c, S_a, D, q_a), da leggersi come "se la macchina si trova nello stato q_c e legge il simbolo S_c, allora scriverà il simbolo S_a, si sposterà in direzione D (sinistra, L, destra, R, o nessuna, N) e passerà allo stato q_a". Turing decise di codificare queste quintuple con numeri naturali secondo queste regole: ad ogni stato q_i si assocerà una stringa composta dalla lettera D seguita dalla lettera A ripetuta i volte, mentre ad ogni simbolo S_j una stringa formata dal carattere D seguito da C ripetuto j volte.

Poiché Turing utilizzava il carattere *;* per separare le quintuple, con la codifica appena introdotta ogni macchina può essere descritta con i soli simboli A, C, D, L, R, N e *;*. Turing chiamò questa rappresentazione una *descrizione standard* ($S.D$). A questo punto è possibile associare ad ogni simbolo un numero intero: da A a *;* si attribuiranno rispettivamente gli interi da 1 a 7, ottenendo così il *numero della descrizione* ($D.N$). A titolo di esempio si riporta la macchina M che produce la sequenza 01010101... descritta da Turing come segue:

$$q_1\ S_0\ S_1\ R\ q_2;\quad q_2\ S_0\ S_0\ R\ q_3;\quad q_3\ S_0\ S_2\ R\ q_4;\quad q_4\ S_0\ S_0\ R\ q_1;$$

dove $S_0 = blank$ (cella vuota), $S_1 = 0$ e $S_2 = 1$.

Applicando le regole della codifica, si ottiene che $q_1 = DA$, $q_2 = DAA$ e così via e, analogamente, $S_0 = D$, $S_1 = DC$ e $S_2 = DCC$. Pertanto, la descrizione standard della macchina M sarà la stringa

$$DADDCRDAA; DAADDRDAAA; DAAADDCCRDAAAA; DAAAADDRDA;$$

e il numero della descrizione sarà

[1] Come riporta lo stesso Turing nel suo articolo, tutte le considerazioni fatte sui numeri computaili sono applicabili anche alle *funzioni computabili*. In altre parole ad una macchina di Turing può essere associata una funzione f *calcolata* dalla macchina stessa. Questo concetto va oggi sotto il nome di *Turing-calcolabilità*.

31332531173113353111731113322531111731111335317.

Si può utilizzare per le macchine di Turing la seguente convenzione[2]: ogni macchina inizia leggendo la prima cifra a sinistra di un numero scritto sul suo nastro. Per alcune di questi numeri la macchina prima o poi si fermerà, mentre per altri può contunuare all'infinito. Si definisce l'insieme dei numeri del primo tipo come l'*insieme di fermata*.

A questo punto è possibile creare un insieme D di numeri naturali, tale che non esiste una macchina di Turing M il cui insieme di fermata sia D. Per ottenere questo risultato verrà usato il metodo della diagonale di Cantor[3] e, per comprenderne meglio il funzionamento, si ricorrerà ad un semplice esempio. Siano $I_1 = \{B, C\}$, $I_2 = \{B, D\}$, $I_3 = \{B, C, D\}$ e $I_4 = \{A, B\}$ degli insiemi di lettere appartenenti ad un alfabeto Σ e sia $f : \Sigma^n \rightarrow \Sigma$ una funzione che crea una corrispondenza biunivoca fra insiemi di lettere e lettere. In particolare, sia $f(I_1) = A$, $f(I_2) = B$, $f(I_3) = C$ e $f(I_4) = D$.

È possibile rappresentare questi insiemi mediante la seguente tabella:

	A	B	C	D
A	⊖	+	+	−
B	−	⊕	−	+
C	−	+	⊕	+
D	+	+	−	⊖

Gli indici delle righe corrispondono alle lettere di Σ, mentre quelli delle colonne sono i valori calcolati da f per i vari insiemi. La tabella è stata costruita nel seguente modo: se un elemento (indice di una riga) appartiene ad un insieme (indice di una colonna), la cella conterrà un $+$, altrimenti un $-$. Si prenda ora in considerazione la diagonale di questa tabella, evidenziata dai simboli cerchiati. È possibile creare una nuova tabella che contenga i valori opposti della diagonale stessa:

	A	B	C	D
	+	−	−	+

Questa tabella corrisponde ad un insieme $I_5 = \{A, D\}$, che ha la proprietà fondamentale di essere diversa da tutte le altre. Infatti, I_5 non può essere l'insieme I_1, poiché questo *non* contiene il simbolo A, contenuto invece da I_5; non può nemmeno essere I_2, in quanto esso contiene B, mentre I_5 no, e così via. Si noti che a prescindere dalla dimensione di Σ o dal numero di insiemi questa proprietà viene sempre rispettata.

Avendo ora chiaro il funzionamento del metodo della diagonale di Cantor è possibile proseguire applicandolo a questo contesto: gli insiemi I_k saranno gli insiemi di fermata descritti sopra e la funzione di corrispondenza biunivoca f sarà definita come $f(IF(M)) = DN(M)$, dove $DN(M)$ e $IF(M)$ sono rispettivamente il numero della descrizione di M e il suo insieme di fermata. A questo punto si può

[2] Questa convenzione, insieme ai risultati proposti da qui in avanti, è ripresa da Il Calcolatore Universale di M. Davis [11].

[3] Il noto metodo della diagonale di Cantor, descritto nel suo articolo *Über eine elementare Frage der Mannigfaltigkeitslehre* [9], permise al suo ideatore, nel 1874, di dimostrare la non enumerabilità dei numeri reali e di stabilire così che $\aleph_0 < C$.

procedere come nell'esempio precedente e costruire D come segue:

$$D = \{DN(M) \mid DN(M) \notin IF(M)\},$$

per ogni macchina M.

In altre parole, se il $D.N$ della macchina M appartiene al suo insieme di fermata, allora non appartiene a D, mentre se lo stesso $D.N$ non appartiene all'insieme di fermata, allora esso apartiene a D. Si è appena costruito, dunque, un insieme che non può essere in alcun caso l'insieme di fermata di M e, poichè questo vale per ogni macchina di Turing, si può concludere che D non è l'insieme di fermata di nessuna macchina di Turing.

Si consideri ora il seguente problema P: definire un algoritmo che, dato un numero naturale n, sia in grado di stabilire se n appartiene o meno a D. Sia per assurdo A l'algoritmo che risolve questo problema: allora esiste una macchina M che implementa A. In particolare si può costruire M in modo che, alla fine della computazione, lasci il nastro completamente vuoto tranne per una cifra che varrà $S_2 = 1$ se $n \in D$ o $S_1 = 0$ se $n \notin D$; inoltre si può fare in modo che M si fermi in uno stato f tale che nessuna delle altre quintuple di M inizi con f stesso. Sia ora M' la macchina costruita aggiungendo a M le due seguenti quintuple:

$$f\ S_1\ S_0\ R\ f;\quad f\ S_0\ S_0\ R\ f.$$

Se il numero in ingresso di M' appartiene a D la nuova macchina si comporterà come M e terminerà con il solo carattere 1 sul nastro, ma se non appartiene a D essa continuerà a spostarsi a destra per sempre. In altre parole, l'insieme di fermata della nuova macchina M' coincide con D. Ma ciò risulta impossibile per costruzione di D, per cui l'ipotesi che esista un algoritmo A in grado di risolvere P deve essere errata e, dunque, tale problema risulta insolubile.

Sia Hilbert, sia Godfrey H. Hardy, matematico britannico, ritenevano che un algoritmo in grado di risolvere l'*Entscheidungsproblem* implicasse la possibilità di decidere algoritmicamente tutti i problemi matematici. Dunque, se esiste un problema matematico algoritmicamente insolubile, allora lo sarà anche il Problema della Decisione. Ma avendo appena descritto un problema P che rispecchia tali requisiti, si può concludere come l'*Entscheidungsproblem* sia algoritmicamente insolubile.

Tesi di Church-Turing La Macchina di Turing, tuttavia, non rappresenta l'unico formalismo in grado di definire la calcolabilità. Negli stessi anni nascevano, ad esempio, il già citato λ-calcolo[4] di Alonzo Church e le funzioni parziali ricorsive di Kleenee e Robinson. Sempre in quel periodo, inoltre, si dimostrava che tutti questi formalismi erano tra loro equivalenti e che, quindi, permettevano di calcolare la stessa classe di funzioni.

Furono proprio questi risultati che, nel '36, portarono Church e Turing a formulare la seguente Tesi fondamentale:

> **Tesi di Church-Turing**: *La classe delle funzioni "intuitivamente calcolabili" coincide con la classe delle funzioni Turing calcolabili.*

[4] Si veda *An Unsolvable Problem of Elementary Number Theory* [10].

Ma cosa si intende per *funzioni intuitivamente calcolabili*? È senza dubbio arduo definire formalmente questo tipo di funzioni, pertanto questa Tesi rimane senza una dimostrazione e non viene chiamata il Teorema di Church-Turing. Tuttavia, accettandola come vera, essa ha delle conseguenze di notevole importanza: le funzioni calcolabili da una Macchina di Turing o da qualsiasi altro formalismo rappresentano effettivamente la classe delle funzioni calcolabili da un algoritmo. In questo modo, dimostrando che una funzione f non è parziale ricorsiva o che non esiste una Macchina di Turing associata a tale funzione, automaticamente si sancirà che f non è calcolabile. In altre parole, la Tesi di Church-Turing permette di limitare l'estensione di ciò che è effettivamente calcolabile.

2.2 Una definizione formale

Finora si è parlato di macchine di Turing in modo piuttosto intuitivo, pertanto si fornirà ora una definizione più rigorosa. Una macchina di Turing (d'ora in poi *MdT*) è una quadrupla $M = (Q, \Sigma, P, q_0)$, dove $Q = \{q_0, ..., q_m\}$ è un insieme finito di stati di cui q_0 è lo stato iniziale, $\Sigma = \{S_0, ..., S_n\}$ è un alfabeto finito che deve contenere almeno due simboli, \$ (*blank*[5]) e 0 (*tally*) e $P = \{I_1, ..., I_p\}$ è un insieme finito non vuoto di istruzioni (*i.e.* le quintuple di cui si è già discusso). Ogni MdT possiede un nastro potenzialmente illimitato suddiviso in celle che contengono ciascuna un carattere sul quale scorre una *testina*.

La testina è intuitivamente il corpo pensante della MdT e può essere definito da un automa a stati finiti (*DFA*[6]). Ogni nodo di questo automa sarà uno stato della MdT e gli archi saranno etichettati con i simboli letti e scritti dalla testina e la direzione da intraprendere. Una rappresentazione alternativa è costituita da una *matrice funzionale*, ovvero una tabella in cui le righe rappresentano gli stati e le colonne i simboli letti dalla testina; in ogni cella della matrice si specificheranno il carattere da scrivere, la direzione in cui deve spostarsi la testina e il nuovo stato della macchina.

La testina così descritta esegue una funzione $\delta : Q \times \Sigma \to Q \times \Sigma \times D = \{L, R, N\}$, detta funzione di transizione, che prende in input uno stato e un carattere e restituisce un nuovo stato, il carattere da scrivere e la direzione. Lo stato in cui si trova la MdT in un certo istante del calcolo è esprimibile mediante il concetto di *descrizione istantanea*, ovvero una quadrupla $ID = \langle q, v, s, w \rangle$, dove q è lo stato attuale della MdT, s è il simbolo letto dalla testina, v e w sono rispettivamente le parti di nastro a sinistra e a destra di s, private delle sequenze illimitate di caratteri *blank*.

Si definisce successore ($\vdash$) di una ID la descrizione istantanea che caratterizza la MdT dopo l'esecuzione di un'istruzione. Grazie a questa nozione, infine, è possibile definire formalmente la computazione di una MdT come una sequenza finita di $ID \alpha_0, ..., \alpha_n$ tali che

$$\begin{cases} \alpha_0 = \langle q_0, v, s, w \rangle \\ \alpha_i \vdash \alpha_{i+1} & \forall i \in \{0, ..., n-1\} \end{cases},$$

dove n è il numero di passi eseguiti dalla MdT. Se $n \geq 0$ e $\alpha_n \nvdash$ (ovvero non esiste un successore di α_n), allora si dice che la computazione è *terminante*.

[5] Il carattere *blank* rappresenta una cella vuota.

[6] Sebbene esistano anche MdT non deterministiche (le cui testine sono definibili tramite *NFA*), in questa sede si tratterà unicamente MdT che non ammettono istruzioni che condividano la stessa configurazione q_c, s_c.

Macchine di Turing a k nastri Quello appena descritto non è che uno dei molti possibili modelli di MdT. Uno dei più comuni è quello delle MdT a k nastri, o k-MdT, definibile come una quintupla $M = (Q, \Sigma, P, q_0, T)$, dove Q, P e q_0 sono definiti come nel modello precedente e $T = \{t_0, ..., t_k\}$ è un insieme finito non vuoto di nastri (ognuno avente la propria testina). Secondo questo modello la funzione di transizione diventa $\delta : Q \times \Sigma^k \rightarrow Q \times \Sigma^k \times D^k$.

L'aggiunta di nastri al modello *standard* di MdT non aumenta tuttavia il numero di funzioni calcolabili mediante una MdT; ciò è valido per qualsiasi altro modello matematico di MdT. Infatti, per MdT definita con un altro modello, è possibile dimostrare che esiste una MdT *standard* in grado di simularla. Nel caso specifico di una k-MdT è possibile ottenere la MdT a un nastro corrispondente con un costo computazionale[7] maggiore.

Ad esempio, si consideri la MdT che copia l'input di dimensione n che ha ricevuto. Nel caso di una macchina M ad un nastro l'input va copiato a destra un carattere per volta: la testina dovrà muoversi di n caselle a destra per copiare e tornare indietro di altrettante caselle per leggere il carattere successivo. Poichè questa operazione viene eseguita n volte, si ha che la complessità di M è $\Theta(n^2)$. Sia ora M_k la MdT a due nastri che esegue la stessa operazione; M_k non è costretta a copiare l'input sullo stesso nastro, ma può farlo sul secondo. In questo modo il processo di copia richiede una sola operazione per carattere e, essendovi n caratteri, la sua complessità sarà pari a $\Theta(n)$.

Macchine di Input/Output Una k-MdT è detta di Input/Output (I/O) se l'insieme T contiene due ulteriori nastri t_I e t_O detti rispettivamente nastro di input e di output. La peculiarità di questi due nastri risiede nelle limitazioni che impongono alla funzione di transizione δ: in un nastro di input non si può sovrascrivere un simbolo con s_i con s_j per nessun $i \neq j$, mentre in un nastro di output è possibile spostarsi solo a destra.

L'idea che sta alla base di queste regole sta proprio nella definizione di *input* e *output*: il primo dev'essere, per fare un'analogia con un file, in modalità *read only*, mentre il secondo in modalità *write only*. Gli eventuali rimanenti nastri possono essere usati per effettuare calcoli intermedi.

Sintassi di MdT Finora si sono descritti modelli di MdT senza però definire delle sintassi e, come per i modelli, esistono diverse sintassi. Ad esempio, l'insieme delle direzioni D presente nelle quintuple (istruzioni) *standard* prevede $D = \{L, R\}$, ovvero sinistra e destra; per comodità è possibile aggiungere un terzo elemento S, che rappresenta il movimento nullo. Nuovamente, si sottolinea che l'aggiunta di questa possibilità non amplia le possibilità di calcolo di una MdT, infatti un movimento S è simulabile da due movimenti consecutivi L e R o viceversa.

Verrà ora descritta, a titolo di esempio, la sintassi utilizzata da Christos H. Papadimitriou nel suo libro *Computational Complexity* [16]. Ogni MdT (a uno o più nastri), avrà un insieme di stati Q che conterrà anche gli stati particolari h (*halting state*, stato di arresto), *yes* (stato di accettazione) e *no* (stato di rifiuto); essi possono essere interpretati come stati finali in cui si finisce, rispettivamente, se vi è un errore, se si ottiene un esito positivo o uno negativo.

[7] Il costo computazionale di una MdT che prende un input di dimensione n consiste nel numero di operazioni che la macchina esegue, espresso da una funzione del tipo $f(n)$. Ad esempio, sia M una macchina ad un nastro che scandisca l'input da sinistra a destra e viceversa per poi fermarsi quando incontra un carattere *first*: con un input di dimensione n il costo computazionale di un'esecuzione di M sarà $f(n) = 2n$.

L'alfabeto Σ, invece, deve contenere oltre al simbolo *blank* il simbolo *first* ($\triangleright$), che ha come vincolo il fatto di obbligare la testina ad andare a destra. Le istruzioni sono quintuple della forma $I = (q, s, q', s', d)$ per le macchine ad un nastro e $(2 + 3k)$-ple della forma $I_k = (q, s_0, ..., s_k, q', s'_0, ..., s'_k, d_0, ..., d_k)$. Inoltre, come convenzione, Papadimitriou impone che ogni nastro cominci con il carattere *first*.

2.3 Simulatori esistenti

Esistono svariati simulatori di MdT e se ne citeranno qui alcuni. Alcuni risalgono ai tempi del DOS, come `TURING.EXE` [4], sviluppato dal *John Kennedy Mathematics Department* del *Santa Monica College*, che consentiva la simulazione di MdT a un nastro tramite un'interfaccia a carattere.

Più recentemente (2012) la *SuperUtils* ha rilasciato `Uber Turing Machine` [6], un simulatore per sistemi Windows per 1-MdT, basato su matrice: le istruzioni sono definite tramite una tabella avente per righe gli stati (trattati come interi, con 0 unico stato finale) e per colonne i caratteri dell'alfabeto.

Esistono anche simulatori multi piattaforma, come `Tursi` [5] (di Claus Schätzle, del *AIS* dell'università di Freiburg) e `Alan` [1] (di Daniel Neuber, Oliver Pahl e Dominik Seichter); entrambi sono basati su matrice e permettono la simulazione di MdT ad un nastro. `Tursi` permette la definizione di macchine tramite file che prende in input e ha l'interessante funzione di *debug* delle MdT, grazie ad una cronologia delle istruzioni e a stati utilizzabili come *break point*. `Alan`, invece, permette l'inserimento e la modifica di istruzioni da interfaccia grafica e permette l'inserimento di una documentazione per ogni MdT.

Infine, sempre nell'insieme dei simulatori multi piattaforma, vi sono simulatori basati su DFA come `Tuatara Turing Machine Simulator` [3] (di James Foulds della *University of Waikato*) per 1-MdT e `JFlap`[8] [2] (di Susan H. Rodger, del *Department of Computer Science of Duke University*), che consente la simulazione di k-MdT. Entrambi consentono la creazione del DFA mediante interfaccia grafica, consentendo di organizzare visualmente nodi (gli stati) e archi (le transizioni).

[8] `JFlap` consente, oltre alla simulazione di k-MdT l'utilizzo di automi, grammatiche, espressioni regolari e Pumping Lemma.

3

A-Machine

Si presenta, infine, il simulatore di k-MdT oggetto di questa dissertazione: `A-Machine`. Il nome per il programma in questione è stato ispirato dallo stesso articolo di Turing di cui si è discusso nel capitolo 2.1:

> «*If at each stage the motion of a machine [...] is completely determined by the configuration, we shall call the machine an "automatic machine" (or a-machine).*» [17]

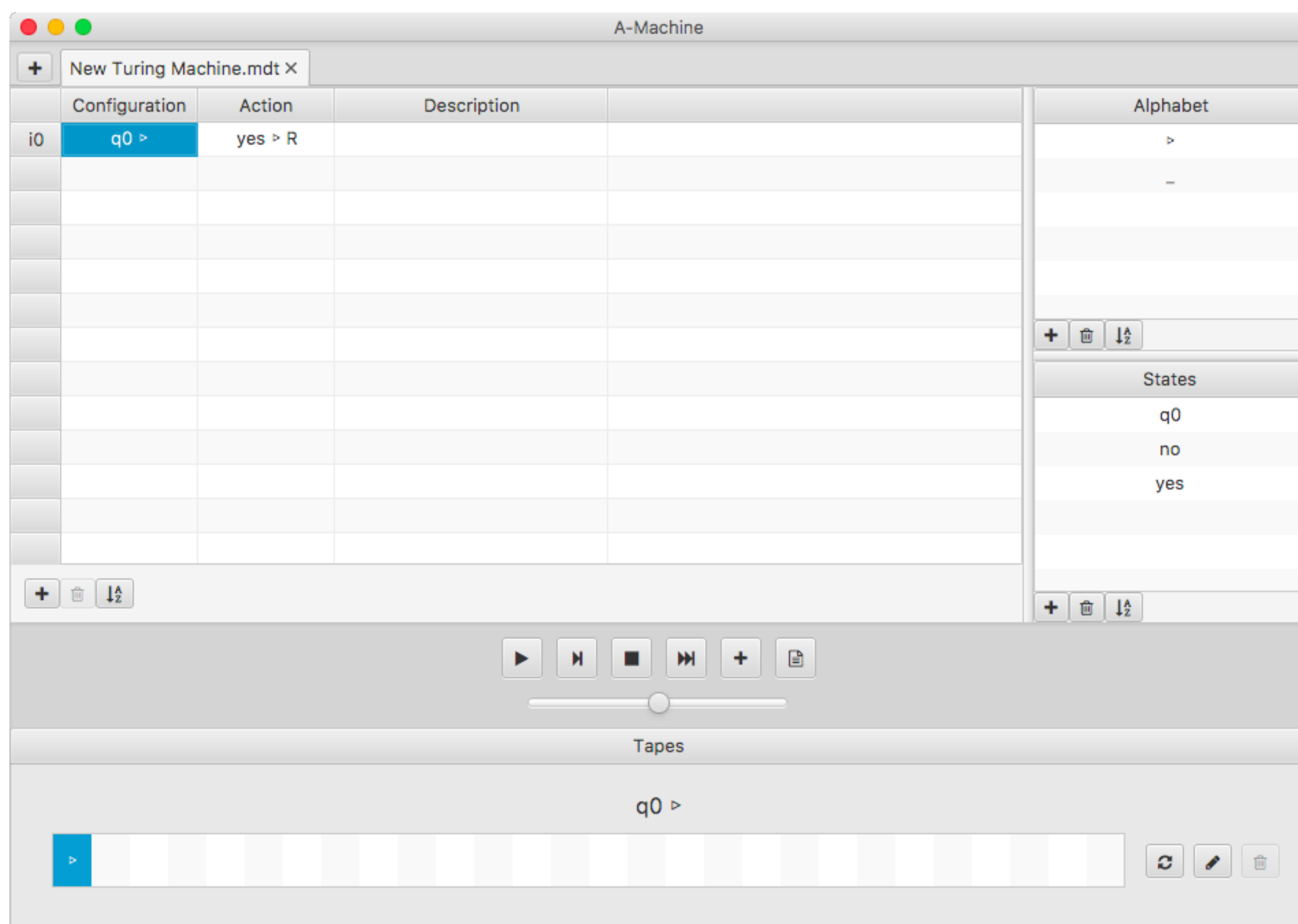

Figura 3.1: Interfaccia grafica di `A-Machine`

Il simulatore è stato realizzato in Java (versione JDK 1.8.0_152-ea), utilizzando la libreria grafica JavaFX e dunque è multi piattaforma.

Alfabeto L'alfabeto è composto da almeno due caratteri, come descritto nella sintassi del Papadimitriou: *first* ($\triangleright$) e *blank* (_). Si possono aggiungere nuovi caratteri, rimuoverli o modificarli, eccezion fatta per *first* (che non può essere modificato o rimosso) e *blank* (che può essere solo modificato).

Stati Ogni stato contiene il proprio nome (che dev'essere univoco) e una descrizione, che può essere utile in fase di progettazione di una MdT. Ogni stato può essere finale, incluso quello iniziale. Aprendo una nuova macchina *vuota*, questa verrà riempita con i caratteri appena descritti e con tre stati: q_0, lo stato iniziale, *yes* e *no*, gli stati finali.

Istruzioni La sintassi adoperata per le istruzioni è quella del Papadimitriou (2.2) e prevede, quindi, l'inserimento di una configurazione e di un'azione corrispondente, che avviene tramite una tabella. Oltre a questi dati, ad ogni istruzione può essere associata una descrizione, che ha lo stesso scopo dei commenti per un linguaggio di programmazione. Una volta inserita, un'istruzione può essere modificata a piacimento o eliminata.

Nastri Ogni nastro contiene i simboli inseriti dall'utente ed è potenzialmente illimitato: l'utente può scorrere a destra e a sinistra e verranno sempre aggiunte nuove celle. È possibile creare, modificare ed eliminare nastri in ogni momento, purché sia presente almeno un nastro. Seguendo la sintassi del Papadimitriou, di default ogni nastro dovrà iniziare con il simbolo *first*; è possibile tuttavia rimuovere questa convenzione dal menu delle impostazioni.

Modificando i valori dei simboli dell'alfabeto e i nomi degli stati, aggiungendo o rimuovendo nastri, il programma aggiornerà automaticamente le istruzioni per renderle coerenti con le modifiche.

Macchine di I/O Con `A-Machine` è possibile utilizzare anche le macchine di input/output (figura 3.2) descritte precedentemente. Alla creazione di una nuova macchina il programma fa scegliere il tipo di MdT utilizzare: una normale k-MdT o una di I/O. Selezionando la seconda opzione l'interfaccia creata conterrà anche i nastri di input e output; sarà comunque possibile eseguire le operazioni sui nastri descritte in precedenza. Aggiungendo nuove istruzioni il programma controllerà che rispettino i vincoli imposti dai nastri di I/O.

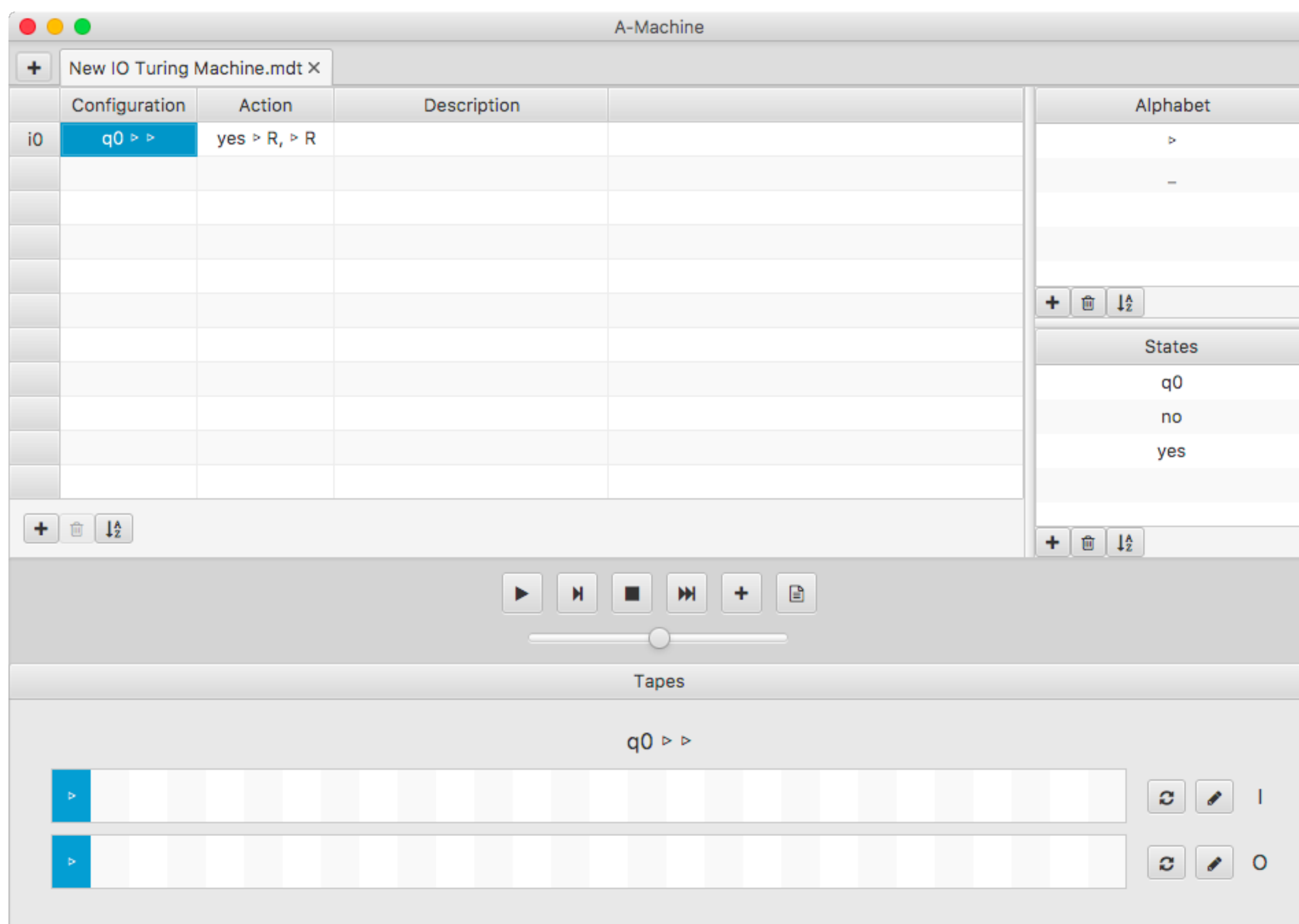

Figura 3.2: Una macchina di I/O

3.1 Macchine esterne

`A-Machine` permette di utilizzare delle macchine esterne all'interno delle istruzioni della MdT che si sta programmando. Queste macchine non sono altro che le *skeleton tables* di cui parlava Turing nel suo articolo[1]:

> «*There are certain types of process used by nearly all machines, and these, in some machines, are used in many connections. These processes include copying down sequences of symbols, comparing sequences, erasing all symbols of a given form, etc. Where such processes are concerned we can abbreviate the tables [...] considerably by the use of "skeleton tables".*»

È possibile selezionare una macchina esterna specificando il path del file corrispondente. Grazie a questa funzione è possibile creare anche macchine ricorsive o mutuamente ricorsive. Tuttavia, affinché una macchina M' sia utilizzabile all'interno di una seconda macchina M, si devono garantire i seguenti requisiti di compatibilità:

- Gli alfabeti devono essere uguali: $\Sigma = \Sigma'$.

[1] Si veda Tur37, §4 [17].

- M e M' devono essere dello stesso *tipo*: k-MdT oppure I/O-MdT.

La prima condizione serve per garantire che non vi siano incoerenze nelle istruzioni, in quanto potrebbero esserci alcuni caratteri presenti in Σ', ma non in Σ. La seconda condizione, invece, serve per fare in modo che vengano rispettati i vincoli dei nastri di I/O. Si noti che la compatibilità fra due macchine non è influenzata dal numero dei loro nastri. Inizialmente, tuttavia, vi era una terza condizione che imponeva $k' = k$. In questo modo, quando si chiamava una macchina all'interno di un'altra, il programma trasferiva i nastri dalla MdT chiamante a quella chiamata e, finita la computazione, di nuovo a quella chiamante.

In questo modo, tuttavia, l'utilizzo di macchine esterne era piuttosto limitato. Si consideri la macchina M_C, avente $k_{M_C} = 2$, che copia il contenuto del primo nastro nel secondo. Ponendo $k' = k$ si ha che ogni macchina che intende usare M_C deve necessariamente avere due nastri; pertanto, se una macchina M avesse $k = 3$ e ad un certo punto dovesse copiare il contenuto del primo nastro nel terzo, non potrebbe utilizzare M_C, ma dovrebbe utilizzare una seconda macchina a tre nastri che faccia la stessa identica cosa. Se poi M avesse bisogno di copiare il secondo nastro nel primo, allora si dovrebbe costruire una terza macchina per la copia. Per questo motivo è stata introdotta la configurazione dei nastri: richiamando una macchina esterna si dovrà specificare una corrispondenza fra i nastri della MdT chiamante e quella chiamata.

Perciò, alla prima chiamata di M_c da parte di M, la configurazione sarà definita dalle coppie $(1,1),(3,2)$, indicando una corrispondenza fra il primo nastro di M e il primo di M_C e fra il terzo nastro di M e il secondo di M_C. Analogamente, alla seconda chiamata la configurazine sarà definita dalle coppie $(2,1),(1,2)$. L'unico vincolo che si impone ad una configurazione è che non vengano associati a più nastri della MdT chiamante uno stesso nastro di quella chiamata: ad esempio, la configurazione $(2,1),(3,2),(1,1)$ non sarebbe valida, poiché il primo nastro della macchina esterna viene associato sia al secondo, sia al primo nastro della MdT chiamante.

Il motivo per cui questa condizione è necessaria sarà chiaro con il seguente esempio. Sia M_I ($k_{M_I} = 1$) la macchina *identità*, che termina immediatamente senza modificare il contenuto del suo nastro. Sia poi M' una macchina a due nastri che richiami M_I con la configurazione $(1,1),(2,1)$. Si può notare subito che si viene a creare un'ambiguità su quale nastro effettivamente debba utilizzare M_I per i suoi calcoli; ma anche potendo risolvere questa ambiguità rimarrebbe un ulteriore problema. Si supponga (senza perdita di generalità) che M_I usi il primo nastro di M'. In questo modo, al termine della computazione di M_I la macchina M' avrebbe nei suoi due nastri il contenuto dell'unico nastro di M_I, ovvero avrà copiato il contenuto del suo primo nastro nel secondo. Tuttavia l'operazione di copia non sarebbe dovuta alle operazioni eseguite dalla macchina M_I (che effettivamente non ne compie), né tantomeno da M: la copia verrebbe effettuata dal programma senza che vi sia necessità di creare una MdT che la esegua.

Se si sostituissero le istruzioni contenute in M_I al posto dell'istruzione di M' che effettua la chiamata si noterebbe un comportamento complessivo di M' del tutto diverso. Inoltre si renderebbero le MdT più *efficienti* (i passi per eseguire la copia non verrebbero compiuti dalle MdT) di quello che effettivamente sono.

3.2 Altre funzioni

Con `A-Machine` è possibile aprire più macchine contemporaneamente, ognuna delle quali viene aperta in una scheda (*tab*), consentendo di lavorare su più file allo stesso tempo. Per verificare il comportamento della macchina su cui si sta lavorando ci sono tre opzioni: il tasto *Run* consente di eseguire le istruzioni in automatico con una velocità selezionabile dall'utente; il tasto *Step* permette di eseguire un'operazione per volta; il tasto *Compute All* esegue interamente la computazione senza pause.

L'aggiunta di questa terza opzione velocizza molto la computazione, ma è vulnerabile a quelle MdT che non terminano mai. Per questo motivo in `A-Machine` è stato introdotto un contatore del numero di istruzioni svolte: quando si raggiunge il valore massimo il programma interrompe ogni MdT in esecuzione e notifica l'errore di *overflow*. Inizialmente il limite superiore è impostato a 1000, ma l'utente può modificarlo dal menu delle impostazioni. Questa funzione di controllo è utile soprattutto per le macchine ricorsive o mutuamente ricorsive.

Ad ogni computazione, inoltre, viene generato un *log* in cui vi sono scritti tutti i passi della MdT in uso, più eventuali operazioni svolte da macchine esterne chiamate dalla MdT in uso. Alla fine del *log* sono inserite le informazioni sul numero di passi totali eseguiti (inclusi quelli delle MdT esterne) e delle celle occupate, per fornire una stima della complessità sia in termini di tempo, sia di spazio.

Infine, `A-Machine` è disponibile sia in lingua inglese, sia in italiano ed è possibile aggiungerne altre modificando il file `Translations.java`, che contiene, per ogni stringa che compare nel programma, un array di traduzioni. Si può selezionare la lingua che si preferisce dal menu impostazioni: in questo modo si modifica il valore dell'indice corrispondente ad una determinata lingua (0 per l'inglese, 1 per l'italiano).

Bibliografia

[1] Alan. `http://alan.sourceforge.net/`.

[2] Jflap. `http://www.jflap.org/`.

[3] Tuatara Turing Machine Simulator. `https://sourceforge.net/projects/tuataratmsim/`.

[4] Turing.exe. `http://archives.math.utk.edu/software/msdos/miscellaneous/jkturing/`.

[5] Tursi. `http://ais.informatik.uni-freiburg.de/tursi/`.

[6] Uber Turing Machine. `http://www.superutils.com/products/uber-turing-machine/`.

[7] Bertrand Russell Alfred North Whitehead. *Principia Mathematica*. Cambridge University Press, 1910.

[8] George Boole. *The Mathematical Analysis of Logic*. Macmillan, 1847.

[9] Georg Cantor. Über eine Elementare Frage der Mannigfaltigkeitslehre. *Jahresbericht Deutschen Math.-Verein.*, 1:75–78, 1891.

[10] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[11] Martin Davis. *Il Calcolatore Universale*. Adelphi, 2003.

[12] Gottlob Frege. *Begriffsschrift*. Verlag von Louis Nebert, 1879.

[13] Gottlob Frege. *Die Grundlagen der Arithmetik*. Verlag von Wilhelm Koebner, 1884.

[14] Gottlob Frege. *Grundgesetze der Arithmetik*. Verlag Hermann Pohle, 1893.

[15] Kurt Gödel. Über Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[16] Christos M. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[17] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.