Università degli Studi di Milano Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di laurea in Informatica

Convertitore di formati di input per Celluloid

Relatore: *Della Vedova Gianluca*
Co-relatore: *Ciccolella Simone*

Relazione della prova finale di:
Fumagalli Danilo
Matricola 830683

Anno Accademico 2019-2020

Contents

Chapter 1

parser e grammatiche

1.1 Introduzione

Esistono tre tipi generali di parser per le grammatiche: universali, top-down e bottom-up. I metodi di parsing universali come l'algoritmo Cocke-Younger-Kasami e l'algoritmo di Earley possono analizzare qualsiasi grammatica, ma sono molto inefficienti.

I metodi comunemente usati possono essere classificati come top-down o bottom-up. come sottointeso dai nomi, i metodi top-down creano parse tree partendo dalla radice (top) alle foglie (bottom), mentre i metodi bottom up iniziano dalle foglie e si fanno strada fino alla radice. In ogni caso, l'input del parser viene scansionato da sinistra a destra, un simbolo alla volta.

I metodi top-down e bottom-up più efficienti funzionano solo per una sottoclasse di grammatiche, ma la maggior parte di queste classi, in particolare, Le gramamtiche LL ed LR, sono abbastanza espressive da descrivere la maggior parte dei costrutti utilizzati nei linguaggi di programmazione. I parser implementati a mano spesso usano grammatiche LL, mentre i parser per la classe più grande delle grammatiche LR sono solitamente costruiti usando strumenti automatizzati.[1]

Per questo progetto ho utilizzato un metodo di parsing detto *packrat parsing*, che mantiene la capacità di riconoscimento di linguaggi di un parser LR, con la semplicità di un parser top-down, utilizzando però una grande quantità di spazio in memoria.

1.2 Context-free grammars

una context-free grammar è formata da simboli terminali, simboli non terminali, un simbolo di partenza, e regole di produzione.[1]

1. I *terminali*, a volte chiamati "token". sono i simboli di base da cui vengono formate le stringhe.

2. I *non terminali*, a volte chiamati "variabili sintattiche". sono le variabili sintattiche che rappresentano insiemi di stringhe

3. Un insieme di *produzioni*, dove ogni produzione consiste in un nonterminale, chiamato la *testa* o il *lato sinistro* della produzione, una freccia, ed una sequenza di terminali e/o nonterminali, chiamata il *corpo* o *lato destro* della produzione. l'obiettivo intuitivo di una produzione è di specificare una delle forme scritte di un costrutto; se il nonterminale di testa rappresenta un costrutto, allora il corpo rappresenta una forma scritta del costrutto.

4. La designazione di uno dei nonterminali come simbolo di partenza.

Scriviamo le grammatiche elencando le loro produzioni, con le produzioni per il simbolo di partenza per prime. Assumiamo che cifre, segni come $<$ e $<=$, e stringhe in grassetto come **while** sono terminali. Un nome in italico è un nonterminale, e qualsiasi nome non in italico o simbolo può essere considerato un terminale.

Esempio: Espressioni formate da cifre e da segni meno e più, per es. stringhe come $9-5+2$, $3-1$, o 7. La seguente grammatica descrive la sintassi di queste espressioni. le produzioni sono:

```
list → list + digit
list → list - digit
list → digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

1.3 FIRST e FOLLOW

La costruzione sia dei parser bottom-up che di quelli top-down è facilitata da due funzioni, FIRST e FOLLOW, associate con una grammatica G. Durante il parsing top-down, FIRST e FOLLOW ci permettono di scegliere quale produzione applicare, in base al prossimo simbolo di input.[1]

Definiamo $FIRST(\alpha)$, dove α è qualsiasi stringa di simboli della grammatica, come l'insieme di terminali che iniziano le stringhe derivate da α. Se $\alpha \stackrel{*}{\Rightarrow} \epsilon$, allora anche ϵ è in FIRST(α)

Per un'anteprima di come FIRST può essere usato durante il parsing predittivo, considera due produzioni $A \rightarrow \alpha \,|\, \beta$, dove FIRST($\alpha$) e FIRST($\beta$) sono insiemi disgiunti. Possiamo scegliere fra queste produzioni guardando

il prossimo simbolo di input a, in quanto a può essere al più in uno fra FIRST(α) e FIRST(β), e non in entrambi. Per esempio, se a è in FIRST(α) scegli la produzione $A \rightarrow \alpha$.

Definiamo FOLLOW(A), per un nonterminale A, come l'insieme di terminali a che possono apparire immediatamente a destra di A in una qualche forma sentenziale; ovvero, l'insieme dei terminali a tali per cui esiste una derivazione della forma $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, per qualche α e β. Nota che potrebbero esserci stati simboli fra A ed a, in un qualche momento durante la derivazione, ma se così fosse, hanno derivato ϵ e sono scomparsi. Inoltre, se A può essere il simbolo più a destra in qualche forma sentenziale, allora \$, simbolo di "fine input" che non è simbolo per nessuna grammatica, è in FOLLOW(A).

1.4 LL(1) grammars

I parser predittivi, ovvero i parser a discesa ricorsiva senza backtracking, possono essere costruiti per una classe di grammatiche chiamata LL(1). La prima L in LL(1) sta per scansione dell'input da sinistra a destra, la seconda "L" per la produzione di una leftmost derivation, e l' "1" per l'utilizzo di un simbolo di input di lookahead in ogni passoper compiere decisioni di parsing.

La classe delle grammatiche LL(1) è abbastanza ricca da coprire la maggior parte dei costrutti di programmazione, anche se è necessaria attenzione nella scrittura di una grammatica adeguata per il linguaggio sorgente. Per esempio, nessuna grammatica left-recursive o ambigua può essere LL(1).

Una grammatica G è LL(1) se e solo se ogniqualvolta $A \rightarrow \alpha \mid \beta$ sono due produzioni distinte di G,valgono le seguenti condizioni:

1. Per nessun terminale a sia α che β derivano stringhe che iniziano con a.

2. Al più uno fra α e β può derivare la stringa vuota.

3. Se β deriva in zero o più passi ϵ, allora α non deriva nessuna stringa che inizia con un terminale in FOLLOW(A).

Le prime due condizioni sono equivalenti ad affermare che FIRST(α) e FIRST(β) sono insiemi disgiunti. La terza condizione è equivalente a dichiarare che se ϵ è in FIRST(β), allora FIRST(α) e FOLLOW(A) sono insiemi disgiunti, e in modo analogo se ϵ è in FIRST(α).

I parser predittivi possono essere costruiti per grammatiche LL(1) in quanto la produzione corretta da applicare per un nonterminale può essere

selezionata guardando solo al simbolo di input corrente. Costrutti di controllo del flusso, con le loro parole chiave distinte, generalmente soddisfano i vincoli LL(1). Per esempio, se abbiamo le produzioni

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\,(\,expr\,)\,stmt\,\textbf{else}\,stmt \\ & |\,\textbf{while}\,(\,expr\,)\,stmt \\ & |\,\{\,stmt\_list\,\} \end{aligned}$$

allora le parole chiave **if**, **while** ed il simbolo { ci dicono quale alternativa è l' unica che potrebbe avere successo se trovassimo uno statement.

1.5 LR parsers

il tipo più diffuso di bottom-up parser al giorno d'oggi si basa su un concetto chiamato parsing LR(k); la "L" sta per scansione left-to-right dell' input, la "R" per la costruzione di una derivazione rightmost invertita, e il k per il numero di simboli di input di lookahead che vengono usati per compiere decisioni di parsing.

I parser LR sono guidati da tabelle, similarmente ai parser LL non ricorsivi. Una grammatica per cui può essere costruita una tabella di parsing LR è detta *grammatica LR*.

Il parsing LR è attraente per una varietà di ragioni:

- Dei parser LR possono essere costruiti per riconoscere virtualmente ogni costrutto di linguaggi di programmazione per cui può essere scritta una grammatica context-free. Esistono delle grammatiche context-free che non sono LR, ma queste possono generalmente essere evitate per i costrutti tipici dei linguaggi di programmazione

- Il metodo di parsing LR è il metodo di parsing shift-reduce senza backtracking più generale conosciuto, eppure può essere implementato tanto efficientemente quanto altri metodi shift-reduce più primitivi.

- Un parser LR può individuare un errore sintattico appena è possibile farlo in una scansione da sinistra a destra dell' input.

- La classe di grammatiche che può essere analizzata utilizzando metodi LR è un sovrainsieme proprio della classe di grammatiche che possono essere analizzate con metodi predittivi o LL. Perchè una gramamtica sia LR(k), dobbiamo essere in grado di riconoscere la presenza del lato

destro di una produzione in una forma sentenziale a destra, con k simboli di input di lookahead. Questa richiesta è molto meno stretta di quella per le grammatiche LL(k) dove dobbiamo essere in grado di riconoscere l'uso di una produzione guardando solo i primi k simboli di ciò che il suo lato destro deriva. Quindi, non dovrebbe sorprendere che le grammatiche LR possono descrivere più linguaggi delle grammatiche LL.

Lo svantaggio principale del metodo LR è che troppo impegnativo costruire un parser LR a mano per la grammatica di un tipico linguaggio di programmazione. Uno strumento specializzato, un generatore di parser LR, è necessario. Fortunatamente, sono disponibili molti generatori di questo tipo. Questi generatori prendono grammatiche context-free e producono automaticamente un parser per quella grammatica. Se la grammatica contiene ambiguità o altri costrutti che sono difficili da analizzare in una scansione left-to-right dell' input, il generatore di parser individua questi costrutti e fornisce messaggi di diagnostica dettagliati.

1.6 PEG

Le *Parsing Expression Grammars* [2] sono stilisticamente simili alle grammatiche context-free, con l'aggiunta di caratteristiche simili alle espressioni regolari. Una differenza chiave è che invece dell' operatore di scelta non ordinata '|', usato per indicare alternative per l'espansione di un non terminale in EBNF, le PEG utilizzano un operatore a scelta *prioritizzata* '/'. questo operatore elenca modelli alternativi da testare *in ordine*, utilizzando il primo match che ha successo. le regole in EBNF 'A $\rightarrow$ a b |a' ed 'A $\rightarrow$ a |a b' sono equivalenti in una CFG, ma le regole PEG 'A $\leftarrow$ a b / a' and 'A $\leftarrow$ a / a b' sono diverse. La seconda alternativa nella seconda regola PEG non avrà mai successo, in quanto la prima scelta viene sempre presa se la stringa di input da riconoscere inizia con 'a'.

definizione: Una *parsing expression grammar* (PEG) è una 4-upla $G = (V_N, V_T, R, e_S)$, dove V_N è un insieme finito di simboli nonterminali, R è un insieme finito di regole, e_S è una parsing expression detta *start expression*, e $V_N \bigcap V_T = \emptyset$. Ogni regola $r \in R$ è una coppia (A, e), che scriviamo $A \leftarrow e$, dove $A \in V_N$ ed e è una parsing expression. Per ogni nonterminale A, esiste esattamente una e tale che $A \leftarrow e \in R$. R è quindi una funzione da nonterminali ad espressioni, e scriviamo $R(A)$ per indicare l'unica espressione e tale che $A \leftarrow e \in R$.

Definiamo le *parsing expressions* induttivamente come segue. Se e, e_1, ed e_2 sono parsing expression, allora lo sono anche:

1. ε, la stringa vuota
2. a, qualsiasi terminale, dove $a \in V_T$.
3. A, qualsiasi nonterminale, dove $A \in V_N$.
4. $e_1 e_2$, una sequenza.
5. e_1 / e_2, una scelta con priorità.
6. e^*, zero-o-più ripetizioni.
7. $!e$, un predicato di negazione.

Questa sintassi astratta non include classi di caratteri, la costante "qualsiasi carattere" '.', l'operatore opzionale '?', l'operatore una-o-più-ripetizioni '+', o l' operatore di and '&', i quali appaiono nella sintassi concreta. Queste caratteristiche della sintassi concreta possono essere sostituite localmente nel modo seguente:

- Consideriamo l' espressione '.' nella sintassi concreta come una classe di caratteri contenente tutti i terminali in V_T.
- se $a_1, a_2, ..., a_n$ sono tutti i terminali indicati in una classe di caratteri nella sintassi concreta, allora consideriamo la classe di caratteri come l' espressione nella sintassi astratta $a_1/a_2/.../a_n$.
- Consideriamo l'operatore opzionale $e?$ nella sintassi concreta come e/ε.
- Consideriamo l'espressione una-o-più-ripetizioni e^+ come ee^*.
- Consideriamo l' operatore di and $\&e$ come $!(!e)$.

Qui un esempio di una PEG che riconosce formule matematiche che applicano le cinque operazioni di base ad interi non negativi.

```
Expr    ← Sum
Sum     ← Product (('+' / '-') Product)*
Product ← Power (('*' / '/') Power)*
Power   ← Value ('^' Power)?
Value   ← [0-9]+ / '(' Expr ')'
```

1.7 Packrat parser

Il *packrat parsing*[3] è una tecnica di parsing top-down che elude la scelta fra predizione e backtracking. il *packrat parsing* fornisce la semplicità, eleganza e generalità del modello con backtracking, ma elimina il rischio di avere tempi di parsing non lineari, salvando tutti i risultati di parsing intermedi mentre vengono elaborati, ed assicurando che nessun risultato venga valutato più di una volta. Le basi teoriche dell' algoritmo sono state sviluppate nel 1970[4, 5], ma la versione in tempo lineare non è mai stata messa in pratica a causa delle ridotte dimensioni delle memorie dei computer dei tempi. Tuttavia, su macchine moderne il costo in spazio di questo algoritmo è ragionevole per molte applicazioni.

Il packrat parsing è stranamente potente nonostante la sua garanzia di tempo lineare. Un packrat parser può essere facilmente costruito per ogni linguaggio descritto da una grammatica LL(k) o LR(k), oltre che per molti linguaggi che richiedono lookahead illimitato e non sono quindi LR. Questa flessibilità elimina molte delle restrizioni scomode imposte dai generatori di parser del lignaggio di YACC. i pacrat parser sono inoltre molto più semplici da costruire dei parser LR bottom-up, rendendo pratico il costruirli a mano.

Lo svantagggio principale del packrat parsing è il suo consumo di spazio. nonostante il suo caso peggiore asintotico sia lo stesso di algoritmi convenzionali, lineare alla grandezza dell' input, il suo utilizzo di spazio è direttamente proporzionale alla dimensione dell'input invece che alla massima profondità di ricorsione, che potrebbero differire di ordini di grandezza. Tuttavia, per molte applicazioni come per compilatori ottimizzanti moderni, il costo di spazio di un packrat parser è probabilmente non più grande delle fasi di elaborazione successsive. questo costo potrebbe dunque essere uno scambio ragionevole per il potere e la flessibilità di parsing in tempo lineare con lookaehead illimitato.

1.8 TatSu

TatSu è uno strumento che prende grammatiche in una variante di EBNF in input, e da in output un packrat parser per PEG in Python.

TatSu può compilare una grammatica contenuta in una stringa in un oggetto *tatsu.grammars.Grammar* che può essere usato per analizzare qualsiasi input, similmente a come il modulo *re* viene usato con le espressioni regolari, o può generare un modulo Python che implementa il parser.

TatSu supporta le regole di ricorsività a sinistra nelle grammatiche PEG, e rispetta l' associatività a sinistra nei parse tree risultanti.

Abbiamo utilizzato *TatSu* per generare i parser per i tre formati di input su cui stiamo lavorando, in modo da avere dei parser efficaci a partire dalle grammatiche relativamente semplici dei formati.

Chapter 2

formati di input

2.1 SASC

Simulated Annealing Single Cell inference (SASC)[6] è un nuovo modello e una struttura robusta basata sul Simulated Annealing per l' inferenza della progressione del cancro da dati SCS. L'obiettivo principale è di superare le limitazioni dell'Ipotesi di Siti Infiniti introducendo una versione del *k-Dollo parsimony model* che permette la cancellazione di mutazioni dalla storia evolutiva del tumore.

Il file di input previsto è un file contenente una matrice ternaria dove le righe rappresentano le cellule e le colonne le mutazioni. Ogni cellula deve essere separata da uno spazio o da un tab. ogni cella della matrice può essere:

$I[i, j] = 0$	la mutazione j non è stata osservata nella cella i
$I[i, j] = 1$	la mutazione j è stata osservata nella cella i
$I[i, j] = 2$	non c'è informazione per la mutazione j nella cella i

esempio:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

2.2 SCITE

SCITE[7] è un pacchetto software per calcolare la storia mutazionale di cellule somatiche. Dati profili di mutazione rumorosi di singole cellule, SCITE

esegue una ricerca stocastica per trovare l'albero di Massima Verosimiglianza (ML) o di Maximum aposteriori (MAP) e/o di campionare dalla distribuzione di probabilità a posteriori. La ricostruzione dell' albero può essere combinata con una stima dei tassi di errore nei profili di mutazione.

SCITE è progettato in particolare per ricostruire la storia mutazionale dei tumori basandosi sui profili di mutazioni ottenuti da esperimenti di sequenziamento a singola cellula degli esoni, ma è in linea di principio applicabile ad ogni tipo di profilo di mutazione (rumoroso) per cui l'Ipotesi di Siti Infiniti può essere fatta.

Il file di input è una matrice di mutazioni, dove ogni colonna rappresenta il profilo di mutazione di una singola cellula, ed ogni riga rappresenta una mutazione. Le colonne sono separate da un carattere di whitespace.

(a) solo l' assenza/presenza della mutazione viene distinta

$I[i, j] = 0$	la mutazione i non è stata osservata nella cella j
$I[i, j] = 1$	la mutazione i è stata osservata nella cella j
$I[i, j] = 3$	non c'è informazione per la mutazione i nella cella j

(b) mutazioni eterozigote ed omozigote vengono distinte

$I[i, j] = 0$	la mutazione i non è stata osservata nella cella j
$I[i, j] = 1$	la mutazione eterozigota i è stata osservata nella cella j
$I[i, j] = 2$	la mutazione omozigota i è stata osservata nella cella j
$I[i, j] = 3$	non c'è informazione per la mutazione i nella cella j

esempio:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

2.3 SPhyR

SPhyR[8] è un algoritmo per ricostruire alberi filogenetici dai dati di sequenziamento a singola cellula. *SPhyR* usa il modello di filogenesi k-Dollo, dove ogni SNV può essere guadagnata una volta ma persa k volte.

Il file di input di SPhyR è testuale. La prima linea elenca il numero di taxa (cellule), seguito dal numero di caratteri (SNVs) nella seconda linea. Ogni linea successiva definisce il valore di ogni carattere per ogni taxon. Più specificamente, i valori ammissibili sono 0, 1 e -1, dove 0 indica l'assenza

della mutazione, 1 indica la presenza della mutazione e -1 indica un dato mancante.

I$[i, j] = 0$	la mutazione j non è stata osservata nella cella i
I$[i, j] = 1$	la mutazione j è stata osservata nella cella i
I$[i, j]$ = -1	non c'è informazione per la mutazione j nella cella i

esempio:

6
5
0 0 0 0 0
0 1 0 0 0
0 0 0 -1 0
0 0 1 1 0
0 0 0 0 1
0 0 0 0 0

Chapter 3

conversione fra i formati

3.1 similarità fra i formati

SASC, SCITE e SPhyR utilizzano in input un dataset SCS, con piccole variazioni fra l' input richiesto dai tre programmi. infatti in tutti e tre i formati:

- il file è una matrice ternaria, che rappresenta la presenza di mutazioni nelle cellule
- 1 rappresenta la presenza della mutazione
- 0 rappresenta l'assenza della mutazione

3.2 differenze fra i formati

I tre formati di input indicano con un carattere diverso la mancanza di informazione per la combinazione di cellula e mutazione corrispondente, ovvero:

- 2 per SASC
- 3 per SCITE
- -1 per SPhyR

diversamente dagli altri due formati, SCITE utilizza le righe della matrice per rappresentare le mutazioni, e le colonne per rappresentare le cellule.

SPhyR richiede che le prime due righe del file comunichino il numero di cellule e mutazioni rispettivamente.

3.3 grammatiche

Il primo passo del programma per la conversione fra i formati è stato scrivere la grammatica dei formati sotto forma di PEG ed utilizzare TatSu per generare i parser. Ho scelto di ignorare gli spazi bianchi nel file di input, in quanto non cambiano le informazioni contenute nel file.

Le grammatiche sono scritte nella sintassi di TatSu, che è una leggera variazione rispetto alla EBNF. Tatsu permette inoltre di aggiungere delle direttive nella grammatica, che controllano il comportamento del parser generato, e sono scritte nella forma `@@name :: <value>`. in tutte e tre le grammatiche è presente la direttiva grammar, che specifica il nome della grammatica, e la direttiva whitespace, che definisce quali caratteri ignorare perchè spazi bianchi (in questo caso vengono ignorati gli spazi bianchi che non sono \n). Le grammatiche così ottenute sono le seguenti:

3.3.1 SASC

```
@@grammar :: SASC
@@whitespace ::  /(?s)[ \t\r\f\v]+/
start = file $ ;
cell = "0" ~ | "1" ~ | "2" ~ ;
row = {cell};
file = ("\n").{ row } ;
```

- la regola *start* matcha *file* e il simbolo di *end of text*, $.
- la regola *file* matcha quante più *row* possibili, suddivise da \n.
- la regola *row* matcha quante più *cell* possibili.
- la regola *cell* matcha 0, 1 o 2.

3.3.2 SCITE

```
@@grammar :: SCITE
@@whitespace ::  /(?s)[ \t\r\f\v]+/
start = file $ ;
cell = "0" ~ | "1" ~ | "2" ~ | "3" ~ ;
row = {cell};
file = ("\n").{ row } ;
```

- la regola *start* matcha *file* e il simbolo di *end of text*, $.

- la regola *file* matcha quante più *row* possibili, suddivise da \n.
- la regola *row* matcha quante più *cell* possibili.
- la regola *cell* matcha 0, 1, 2 o 3.

3.3.3 SPhyR

```
@@grammar :: SPHYR
@@whitespace ::  /(?s)[ \t\r\f\v]+/
@@eol_comments :: /#([^\n]*?)$/
start = cell_number_line "\n" snv_number_line "\n" file $ ;
cell_number_line = /\d+/;
snv_number_line = /\d+/;
cell = "0" ~ | "1" ~ | "-1" ~ ;
row = {cell};
file = ("\n").{row};
```

Il formato di input SPhyR include commenti "python style", che iniziano con # e terminano al primo carattere di newline. la direttiva @@eol_comments definisce la struttura dei commenti che il parser deve ignorare.

- la regola *start* matcha in *cell_number_line* e *snv_number_line* le prime due righe del file, ed in *file* la matrice contenente i dati, assicurandosi che il file termini con il simbolo di *end of text*, $.
- *cell_number_line* e *snv_number_line* contengono solo un numero.
- la regola *file* matcha quante più *row* possibili, suddivise da \n.
- la regola *row* matcha quante più *cell* possibili.
- la regola *cell* matcha 0, 1, 2 o 3.

3.4 codice

La maggior parte del codice scritto si trova in **Translator.py**, che usa i moduli generati con TatSu **parserSASC**, **parserSCITE** e **parserSPHYR**.

3.4.1 Translator

Translator.py può essere eseguito individualmente o come parte di celluloid, e richiede i seguenti parametri da linea di comando:

- **inputFormat**, il formato del file da convertire (SASC,SCITE o SPhyR). Se omesso, il default è SASC.
- **outputFormat**, il formato in cui si vuole convertire il file. Se omesso, il default è SASC.
- **outfile**, il file di output. Se omesso, il default è stdout.
- **file**, il file di input.

convert

La funzione *convert* prende come parametro un' oggetto argparse.Namespace contenente le variabili *outputFormat*, *inputFormat*, *outfile* e *file*, che rappresentano i rispettivi parametri da linea di comando descritti sopra.

La funzione legge il file di input, e successivamente chiama *parse_string*, *translate* e *write_file* per analizzare, convertire e scrivere il file nel nuovo formato.

parse_string

La funzione *parse_string* ha due parametri:

- **file_as_string**, il contenuto del file di input sotto forma di stringa.
- **file_format**, il formato (SASC, SCITE, SPhyR) del file di input in una stringa.

L'output della funzione consiste in una lista di liste, dove ogni elemento delle liste più interne è un carattere della matrice contenuta nel file.

translate

La funzione *translate* ha tre parametri:

- **input_ast**, la matrice di caratteri, sotto forma di lista di liste di caratteri.
- **format1**, il formato di partenza.
- **format2**, il formato in cui si vuole tradurre

L'output della funzione è una lista di liste, dove ogni elemento delle liste più interne è un carattere della matrice nel formato di destinazione.

write_file

La funzione *write_file* ha tre parametri:

- **ast_translated**, una lista di liste, contenente i caratteri da scrivere nel nuovo file.
- **file_name**, il nome del file in cui scrivere la matrice.
- **file_format**, il formato del file in cui scrivere la matrice.

La funzione scrive nel file scelto la matrice in input nel formato specificato.

Chapter 4

motivazioni e conclusione

4.1 motivazioni

Una parte della bioinformatica si occupa di ricostruire gli alberi evoluzionari di eventi di mutazione che si suppone abbiano creato un tumore, usando diversi modelli per inferire l' albero evoluzionale da un campione di tumore. Tuttavia, la grandezza e risoluzione dei dataset con cui abbiamo a che fare sono in continuo aumento, portando allo sviluppo di celluloid[9], un nuovo metodo per ridurre la grandezza dei dataset raggruppando delle mutazioni. Il formato di input ed output di celluloid è lo stesso del formato di input di SASC [6], ma si voleva rendere facilmente fruibile celluloid anche ad altri metodi che utilizzano dataset di sequenziamento a singola cellula.

4.2 conclusioni

Lo strumento *convert* per celluloid è in grado di convertire fra loro i formati di dati derivanti dal sequenziamento a singola cellula utilizzati da diversi algoritmi per la ricostruzione di alberi evoluzionari di tumori. Il tool può essere espanso aggiungendo altri formati a quelli supportati, aumentando ulteriormente l'utilità di celluloid nel ridurre le dimensioni dei dataset. L'utilizzo di PEG e packrat parser per la definizione e analisi rispettivamente dei formati permette infatti una facile aggiunta di nuovi formati, con minime modifiche del programma.

Bibliography

[1] "Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman". *"Compilers: Principles, Techniques, and Tools"*. Addison-Wesley, 1986.

[2] Bryan Ford. "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation" .

[3] Bryan Ford. "Packrat Parsing: Simple, Powerful, Lazy, Linear Time" .

[4] "Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman". *"The Theory of Parsing, Translation and Compiling - Vol. I: Parsing."*. Prentice Hall, Englewood Cliffs, N.J., 1972.

[5] Alexander Birman and Jeffrey D. Ullman. "Parsing algorithms with backtrack. Information and Control". 1973.

[6] Simone Ciccolella, Camir Ricketts, Mauricio Soto Gomez, Murray Patterson, Dana Silverbush, Paola Bonizzoni, Iman Hajirasouliha and Gianluca Della Vedova. Inferring cancer progression from single-cell sequencing while allowing mutation losses. 2020.

[7] Katharina Jahn, Jack Kuipers and Niko Beerenwinkel. Tree inference for single-cell data. 2016.

[8] Mohammed El-Kebir. Sphyr: tumor phylogeny estimation from single-cell sequencing data under loss and error. *Bioinformatics*, 34, 2018.

[9] Simone Ciccolella, Murray D Patterson, Paola Bonizzoni, Gianluca Della Vedova. Effective clustering for single cell sequencing cancer data. 2019.