INDICE:

1. Introduzione

Il progetto Tofu, diminutivo di "Toy Functions" language, consiste nello sviluppo di un traduttore dal linguaggio Tofu al linguaggio intermedio, detto T-code, e di un interprete, cioè una macchina virtuale, del linguaggio intermedio.
La traduzione, che rappresenta il front-end del compilatore, si svolge in quattro fasi:

- Analisi lessicale
- Analisi sintattica
- Analisi semantica
- Generazione di codice intermedio (T-code)

L'interprete è un programma che esegue direttamente le istruzioni del T-code, senza compilarle in linguaggio macchina.
Tofu è un linguaggio a paradigma pseudo-funzionale, strutturato in tre sezioni:

- Variabili: è opzionale e introdotta dalla key word *variables*. In questa sezione si dichiarano i nomi e i tipi delle variabili utilizzate nel programma. I tipi atomici disponibili sono int, string e bool, a cui si aggiunge il costruttore di tipo lista.
- Funzioni: è opzionale e introdotta dalla key word *functions*. In questa sezione vengono dichiarate un numero di funzioni arbitrario, ognuna delle quali è caratterizzata da un nome, da zero o più parametri coi rispettivi nomi e tipi, da un tipo ritornato e da un corpo che è costituito da un'espressione senza effetti collaterali.
  Nota: il tipo ritornato void per le funzioni non esiste in tofu.
- Corpo: è obbligatorio e introdotto dalla key word *body*. Tale sezione costituisce il corpo del programma, rappresentato da una sequenza di istruzioni. Un'istruzione corrisponde a un assegnamento di una variabile, possibile solo in questa sezione, o al display del valore di un'espressione.

Le modalità di compilazione dei file sorgenti contenuti nella cartella "*progetto_tofu*" e gli output prodotti sono indicati nel file "*makefile.txt*", tuttavia verranno comunque descritte all'interno di un paragrafo dedicato (8.Compilazione).

Il seguente progetto è pensato per essere compilato ed eseguito tramite terminale Linux. Se l'utente possiede un computer con sistema operativo Linux, allora può saltare la prossima sezione. Altrimenti gli utenti con un altro sistema operativo (es. Windows o MacOS) sono invitati a seguire le istruzioni della seguente sezione.

2. Installazione Ubuntu

Per effettuare l'installazione di Ubuntu seguire le istruzioni indicate alla pagina https://wiki.ubuntu-it.org/Installazione/InstallareUbuntu.
Si prega di fare attenzione durante la scelta delle proprie credenziali (username e password) in quanto non sussistono meccanismi di recupero. Se ci fossero dei problemi dopo l'installazione del software consultare la pagina https://wiki.ubuntu-it.org/Installazione/PostInstallazione
per controllare se viene descritta una soluzione al proprio problema.
Una volta terminato verificare la presenza del percorso \\wsl.localhost\Ubuntu\home\usernamescelto e inserire al suo interno la cartella denominata "*progetto_tofu*" che contiene tutti i file necessari al corretto funzionamento del programma. Quindi aprire il terminale Linux ed eseguire sempre il comando "*cd progetto_tofu*" prima di compiere qualunque azione sui file presenti nella cartella "*progetto_tofu*". Per accertarsi della presenza dei file si può eseguire il comando "*ls*".
Se l'utente utilizza Ubuntu su una macchina che ha Windows come SO, bisogna stare attenti a un problema subdolo: i file .txt di Windows hanno un carattere EoL (End of Line) differente rispetto ai file .txt di Linux. Se si aprono i file presenti in "*progetto_tofu*" in modalità .txt, verificare che in basso a destra ci sia scritto "Unix (LF)" e non "Windows (CRLF)". Se così non fosse o si ha la necessità di creare nuovi file di testo, digitare almeno un carattere di ritorno a capo all'interno del file, dopodiché eseguire il comando "*tr -d '\15\32' < winfile.txt > unixfile.txt*" da terminale.

3. Analisi lessicale

Il principale ruolo dell'analisi lessicale è definire quali stringhe lessicali, cioè sequenze significative di caratteri, possano essere trasformate in simboli, mediante un processo di astrazione, che rappresenteranno gli input dell'analizzatore sintattico. Tale compito è svolto col supporto del generatore di analizzatore lessicale Lex ed è sintetizzato nel file denominato "*lexer.lex*". Lex è strutturato nel seguente modo:

Dichiarazioni
%%
Regole di traduzione
%%
Funzioni ausiliarie

Le "Dichiarazioni" possono essere black box e white box. Quelle black box sono presenti nella forma "%{#include, costanti, variabili%}", mentre quelle white box rappresentano definizioni regolari del tipo "nome expr", che nel file sopra citato rappresentano gli spazi, singoli caratteri denominati "sugar", i commenti, introdotti in ogni riga dal carattere "#", le costanti booleane, intere, stringhe e le variabili, che devono iniziare con almeno una lettera e a seguire può esserci opzionalmente una serie di lettere e/o cifre.

Le "Regole di traduzione" specificano quali stringhe lessicali sono riconosciute dall'analizzatore e quali invece no, causando la terminazione del programma con la segnalazione di un errore e indicando il numero di linea in cui ha origine.

Esempio:

- Il carattere "&" non è presente in Lex, ergo qualora venisse individuato nella quarta riga (per ipotesi) all'interno del file che rappresenta un programma scritto in tofu, esso terminerebbe e l'output sarebbe il seguente:
  Line 4: error on symbol "&".

Inoltre, nella presente sezione accanto ad ogni stringa lessicale viene indicato il valore di ritorno generato da Lex, che rappresenterà l'input dell'analizzatore sintattico. Quindi osservando le "Regole di traduzione" si può ricavare che le operazioni ammesse in tofu sono:

- quelle aritmetiche di addizione (+), sottrazione (-), moltiplicazione (*), divisione (/), fra interi, e negazione (-), applicabili solo agli interi;
- quelle di comparazione (>, >=, <, <=), applicabili ai tipi interi e stringa;
- quelle di uguaglianza (==) e disuguaglianza (!=) applicabili a tutti i tipi;
- quelle logiche (and, or, not) applicabili solo ai booleani;
- quelle applicabili al solo tipo lista di cui solo l'append (++) è specificato in Lex, mentre le rimanenti (empty, length, head, tail) verranno specificate nell'analisi semantica (vedi paragrafo 5).

Nella sezione delle "Funzioni ausiliarie" si possono dichiarare un numero arbitrario di funzioni, anche nessuna, e scritte in linguaggio C, che vengono richiamate nella sezione sovrastante. Se si fosse interessati solo all'analizzatore lessicale, bisognerebbe scrivere qui la funzione main(). In "*lexer.lex*" è presente un'unica funzione ausiliaria: newstring().

4. Analisi sintattica

L'analizzatore sintattico ha come scopo la definizione dell'albero sintattico astratto che potenzialmente può rappresentare la costruzione di qualsiasi programma tofu ammissibile; tuttavia, una volta eseguito un determinato programma, rappresenta la struttura sintattica specifica di quest'ultimo. L'albero è costruito a partire da un nodo radice e collegato a uno o più nodi che sono i suoi "figli". Questi ultimi essendo alla pari vengono considerati "fratelli" e possono essere classificati in due tipologie: non terminali oppure terminali. I nodi non terminali hanno dei figli a loro volta, che rientreranno in una delle due tipologie precedenti (essi sono i figli dei figli della radice, dunque sono i suoi "nipoti). Quindi dalla radice possono estendersi ramificazioni di nodi non terminali con differenti e indefiniti a priori livelli di profondità, fino a quando non si raggiungerà un nodo terminale, detto anche foglia, che, come dice il nome stesso, è privo di figli e conclude la ramificazione. Quando ogni ramificazione ha raggiunto il termine, allora l'albero risulterà completo.
L'identificazione dell'albero astratto è di cruciale importanza poiché sia l'analisi semantica che la generazione di codice intermedio si baseranno su di esso per svolgere i rispettivi compiti.
L'analizzatore lessicale è rappresentato dal file "*parser.*y" e come per l'analisi lessicale, anche in questo caso viene utilizzato uno strumento di supporto: Yacc (Yet Another Compiler Compiler). È un generatore di parser LALR(1) (Look-Ahead Left to Right): un parser di tipo bottom up, dunque dal basso verso l'alto, che guarda solo il prossimo simbolo in input, cioè di lookahead, per decidere quale azione intraprendere e quindi quale regola applicare. Yacc è strutturato esattamente come Lex (vedi paragrafo 3), ma con rilevanti differenze a livello semantico.
Le "Dichiarazioni" white box consistono principalmente nella definizione di token, che nel file "*parser.*y" costituiscono le parole chiave, le operazioni, le costanti, i tipi atomici e i nomi di variabili/funzioni in tofu.
Le "Regole di traduzione" sono in grado di sfruttare la ricorsione senza che ci sia alcuna ambiguità nell'identificazione della regola da applicare e definiscono come generare l'albero astratto. Si presentano nel seguente modo:

```
A : a1 {azione 1}
  | a2 {azione 2}
  …
  | an {azione n}
  ;
```

dove A è un nodo non terminale, il simbolo "|" rappresenta alternative disgiuntive fra loro e ai, con i da 1 a n, rappresenta un frammento di codice in tofu (che può essere anche un singolo nodo non terminale/terminale). In Yacc esistono delle pseudo-variabili per referenziare il valore di attributi semantici: $$ per il nodo non terminale a sinistra (A), $i per l'i-esimo nodo a destra. Ecco allora che {azione i-esima} è un frammento di codice C che delinea le regole di costruzione dell'albero, in cui si sfruttano i puntatori ai nodi p1,p2,p3 definiti nel file "*def.h*" e le "Funzioni ausiliarie" presenti nella terza sezione di Yacc.
Esempio di una "Regola di traduzione":

```
var_section :  VARIABLES var_decl_list  {$$ = nontermnode(NVAR_SECTION);
                 $$->p1 = $2;}
               | {$$ = nontermnode(NVAR_SECTION);}
               ;
```

Nell'esempio la prima alternativa definisce var_section come un nodo non terminale e risulta avere come figlio var_decl_list, mentre la seconda alternativa definisce var_section come un nodo terminale.
Se il programma scritto in tofu risulta violare una delle "Regole di traduzione" quest'ultimo termina con la segnalazione di un errore e indicando il numero di linea in cui ha origine.
Esempi:

- Se una variabile è definita come tipo float, che non esiste in tofu, alla riga x, allora comparirà il seguente messaggio: Line x: error on symbol "float".
- Se si omette l'intera sezione body, cioè il corpo, e x è l'ultima linea di codice, il messaggio sarà il seguente: Line x: error on symbol "".

La funzione ausiliaria parser(), dato che costruisce l'albero astratto, è quella che viene passata all'analizzatore semantico. Essa si occupa di ricevere lo standard input (stdin) e invoca la funzione yyparse() di Yacc, che effettua la vera e propria generazione dell'albero sintattico: se risulta uguale a zero, tutto è andato bene e si procede alla stampa dell'albero (che comunque è stato memorizzato), altrimenti viene invocata la yyerror(), la quale segnala l'errore e termina il programma.

5. Analisi semantica

Si tratta di computare informazione aggiuntiva rispetto all'analisi sintattica, ma che è necessaria per la compilazione, ed in particolare i suoi ruoli principali sono due: la costruzione della Symbol Table e il type checking. La Symbol Table (cioè tabella dei simboli) è appunto una tabella che tiene traccia dei nomi dati a variabili, funzioni e parametri. Invece, il type checking (cioè il controllo sui tipi) consiste nel verificare che non ci siano inconsistenze fra come una cosa è stata dichiarata e come effettivamente viene istanziata; dunque, quali frasi sono semanticamente corrette e quali no. Ad esempio, se "x" è una variabile di tipo intero ed eseguo l'assegnamento "x = true", allora la frase risulta scorretta e verrà segnalata dall'analizzatore semantico, che è il file "*semantics.c*". Dato che per l'analisi semantica non esistono strumenti automatizzati e universali, quali Lex e Yacc, viene di seguito illustrata la logica implementativa usata per scrivere l'analizzatore semantico.
Un elemento della Symbol Table è composto da

- Il nome, che l'elemento ha nel programma tofu;
- La classe a cui appartiene (variabile, funzione o parametro);
- L'object identifier, che lo identifica univocamente all'interno della propria classe tramite un numero;
- Il tipo, che può essere intero, stringa, booleano o lista; nel caso sia una lista, allora conterrà informazioni anche sulla sua profondità, cioè quanto è innestata, e sul tipo atomico della lista: intero, stringa, booleano o sconosciuto, in caso di lista vuota;
- Una Symbol Table locale, utile solo se si tratta di una funzione e differente per ciascuna di esse, all'interno della quale verranno salvati i parametri, vista la loro visibilità locale in tofu, a differenza di variabili e funzioni;
- La struttura Formals, che contiene il numero dei parametri e le loro istanze per ciascuna funzione;
- Il puntatore next, che collega, qualora ce ne fosse la necessità, due elementi che occupano la stessa posizione all'interno della Symbol Table.

In "*semantics.c*" la Symbol Table è un array di 541 elementi (quella locale di 89 elementi), dimensione comunque decidibile in modo arbitrario, in cui la posizione di ciascun elemento è decisa facendo l'hash del suo nome, cioè convertendolo da stringa lessicale a un numero compreso fra 0 e 540 (o fra 0 e 88), tramite a un'apposita funzione denominata proprio per tale ragione hash(). Grazie alla presenza di una struttura come la Symbol Table, diventa più semplice il compito di individuare elementi (variabili/funzioni/parametri) duplicati o usati ma mai dichiarati, tramite la funzione lookup(), ed il loro type checking all'interno di ogni espressione. Le principali funzioni che si occupano di costruire la Symbol Table sono insert(), whereInsert() e whereInsertFunc().

Invece la funzione che si occupa di avviare "*semantics.c*", e che verrà passata al generatore di codice intermedio, è la semantics(), la quale riceve la parser() da Yacc, invoca la createAndCheckSymbolTable() (che d'ora in poi chiameremo cacs()), che è il cuore dell'analizzatore semantico, ed infine si occupa di stampare la Symbol Table sul file "SymbolTable.txt".
La cacs() innanzitutto utilizza la insertBuiltinFunc(), il cui ruolo è quello di inserire all'interno della Symbol Table le funzioni built-in di tofu, cioè la head(), la tail(), la empty() e la length(), che come già anticipato al paragrafo 3, servono per svolgere operazioni su liste. Dopodiché la cacs(), partendo dalla radice, scorre l'albero sintattico dall'alto verso il basso, incontrando quindi per prima la sezione in cui vengono dichiarate le variabili, se esiste, che vengono allocate nella Symbol Table invocando la funzione insertVariable(), la quale effettua anche controlli su eventuali duplicati. Segue la sezione riguardante le funzioni, se esiste, in cui ciclando finché il puntatore non è nullo, ciascuna funzione tofu viene inserita nella Symbol Table tramite la insertFunction(), assieme a tutti i suoi parametri grazie alla insertParameter(), che li colloca nella Symbol Table locale associata e ne passa il numero e le istanze alla struttura Formals; in entrambi i casi con check sui doppioni. Infine, in questa sezione è necessario effettuare il type checking fra il tipo ritornato dichiarato dalla funzione tofu, salvato nella Symbol Table, e il tipo della sua espressione. Per concludere la cacs() effettua la verifica dei tipi degli statement del corpo (in particolare che per gli assegnamenti il tipo di sinistra coincida con quello di destra).
La funzione centrale per il type checking è la exprType(), il cui funzionamento si basa sul meccanismo della ricorsione, che effettua un'uguaglianza fra il valore intero dell'attuale nodo ed il valore di ciascun possibile nodo non terminale o terminale che abbia senso, e quando risulta vera, svolge le specifiche operazioni riguardanti il nodo in questione. La exprType() si avvale di alcune funzioni ausiliarie, fra le quali spicca la typeEqual() che fa il vero e proprio controllo fra due tipi, segnalando un errore se riscontra un'anomalia.
Esempi di errori dell'analizzatore sintattico:

- Se si dichiara una variabile come stringa, ma la si istanzia con un valore intero, viene dato il seguente messaggio:
  "Error: incompatible types; must be all STRING or INT";
- Se due funzioni vengono chiamate allo stesso modo, per esempio "max", l'output prodotto dal programma tofu sarà:
  "Error: identifier "max" is a duplicate function or it has been declared as a variable";

- Se si dichiara una variabile come lista di profondità due, però la si istanzia con profondità uno verrà visualizzato il seguente messaggio:
  "Error: incompatible list types; must both have depth 2 or 1".

6. Generazione di codice intermedio

Partendo dal codice sorgente, cioè il programma tofu, e facendo riferimento all'albero astratto, in questa fase viene generato un programma target, scritto con codice intermedio; in questo caso prende il nome di T-code e il file che lo crea è il "*gen.c*". Il T-code converte ciò che viene descritto dall'albero sintattico tramite l'utilizzo di appositi operatori che ne sintetizzano/identificano il significato. Per esempio, la dichiarazione di una nuova variabile booleana in T-code è espressa dal singolo operatore "NEWB". Il T-code è composto da un insieme di statement, i quali sono caratterizzati da un operatore, un indirizzo, che rappresenta la sua posizione all'interno del codice, da un numero di argomenti variabile che va da zero a due ed infine hanno un puntatore allo statement successivo. L'intero T-code è rappresentato da una struttura che ne identifica la dimensione, cioè numero di istruzioni totali, ed il primo e l'ultimo statement dai quali è composto. L'dea è quella di creare dei singoli segmenti di codice per mezzo delle funzioni makecode(), makecode1(), makecode2() per ciascun operatore (le funzioni citate saranno inovocate da funzioni specializzate per ogni operatore). Dopodiché i segmenti creati verrano concatenati fra loro tramite le apposite funzioni concode() e appcode(): questo procedimento risulta efficace in quanto non bisogna conoscere la lunghezza del T-code o quali operatori verranno usati a priori.
La funzione che avvia il generatore di T-code è la gen(), che riceve la semantics(), invoca la genTcode() e apre/crea il file "IntermediateCode.txt". Si noti che non è di tipo void, ma ritorna il codice T-code, che verrà passato tramite tale funzione all'interprete. Nella genTcode() viene innanzitutto richiamata la genVarBody() che si occupa di creare il codice intermedio riguardante le variabili e di concatenarlo a quello generato per il corpo oppure di creare solo quello del body se la sezione delle variabili non esiste. Quindi si ritorna alla genTcode() per generare il codice riguardante la sezione delle funzioni, se esiste, e lo si concatena a quello generato in precedenza per le variabili e il corpo. Infine, la genTcode() si occupa di scrivere il T-code su file. Bisogna tener presente che la genTcode() si appoggia all'albero astratto per svolgere il proprio compito ed alla funzione ausiliaria genExprCode() per ricavare il codice specifico di ogni operazione possibile in tofu e dei nodi terminali. È in questa funzione che si analizzano le function call, e quindi vengono identificate sono

presenti o meno quelle built-in. Nel T-code ci sono tre istruzioni significative: la prima che è “TCODE size”, in cui la size rappresenta la dimensione del codice, la “VARS num_vars”, dove num_vars è il numero di variabili dichiarate e la “HALT” che è l’ultima istruzione del T-code.

7. Interprete

L’interprete, che è rappresentato dal file “*interpreter.c*”, riceve in input il programma intermedio, scritto in T-code, insieme a eventuali input richiesti all’utente, e genera in uscita l’output del programma scritto in tofu, quindi ciò che serve veramente all’utente.
La sua architettura si basa sull’utilizzo di diversi stack, ognuno riservato per specifici compiti:

- La struttura denominata Tcode, nella quale viene memorizzato il codice intermedio attraverso l’uso dei puntatori; dunque, rappresenta l’intero programma tofu e si tratta di una memoria fissa;
- Lo stack delle variabili in cui vengono memorizzate le loro istanze nella posizione che coincide con il rispettivo oid (object identifier); anche questa è una memoria fissa;
- Lo stack di attivazione dove vengono salvate le chiamate di funzioni; è flessibile, cioè i suoi valori possono essere sovrascritti da valori successivi, in quanto una volta che la funzione termina, non serve tenerla in memoria per sempre;
- Lo stack degli oggetti, all’interno del quale, almeno all’inizio, viene salvata ciascuna istanza del programma tofu, che però può essere sovrascritta perché non è più necessaria oppure viene memorizzata nello stack delle variabili o delle istanze;
- Lo stack delle istanze contiene le istanze delle liste, che possono essere tipi atomici o delle liste a loro volta; ciascuna istanza non viene mai sovrascritta;
- Infine, c’è lo stack dei parametri, che può essere considerato ausiliario, in quanto si occupa semplicemente di salvare i parametri correnti della funzione chiamata, in modo che possano essere caricati quando richiesti, dopodiché verrà sovrascritto dai parametri di un’altra funzione.

Ogni stack ha visibilità globale e, tranne quello dei parametri, hanno un indice globale per poter accedere ai rispettivi valori, il quale sarà incrementato/

decrementato a seconda della necessità. Uno stack è una struttura formata dal tipo (int, bool, string, list), dalla lunghezza, utile per le liste, da un'stanza, che è il valore concreto dell'oggetto, e da un intero denominata "appn", il quale serve per concatenare le liste. Tuttavia, lo stack di attivazione presenta una struttura differente ed è caratterizzato dal numero di parametri della funzione, dalla posizione del primo parametro all'interno dello stack degli oggetti, e dal valore dell'indirizzo del codice a cui ritornare una volta conclusa la chiamata.
La macchina virtuale in realtà non ha un funzionamento complicato: il main() si occupa di invocare la startmachine(), in cui è presente la funzione gen(), e che perciò ha il compito di generare il codice intermedio ed inizializzare gli stack.
Successivamente il main() scorre ogni operatore del T-code, finché non incontra "HALT", e per ciascuno richiama la execute(), che si occupa di eseguire un'azione specifica in base all'operatore identificato, invocando la "exec_op" opportuna.
Si noti che il main() ha due parametri che sono il numero di argomenti passati e le loro istanze. Gli input richiesti all'utente sfruttano questo meccanismo e dunque, come verrà specificato nel paragrafo 9, devono essere forniti subito accanto al nome dell'eseguibile linux e prima del file di input del programma tofu.
Lo stack delle istanze prevede che, se ad esempio nel programma tofu è presente la lista "[[true],[false]]" e supponendo che il suo indice sia zero, allora:

- In zero sia salvata l'istanza true
- In uno sia salvata l'istanza false
- In due è salvata una lista con istanza zero (riferimento a true memorizzato in zero)
- In tre sia salvata una lista con istanza uno (riferimento a false memorizzato in uno)

Nello stack degli oggetti sarà presente una lista di lunghezza due e la cui istanza sarà due.
Invece se si vuole svolgere un'operazione di append, come "[[[12,13]]++[[14]]]", la logica dello stack delle istanze, partendo da zero, è la seguente:

- In zero viene salvato 12
- In uno viene salvato 13
- In due viene salvata la lista con istanza zero
- In tre viene salvato 14 (il valore del suo "appn" è quattro)
- In quattro viene salvata la lista con istanza tre
- In cinque la lista con istanza due

In questo modo quando l'interprete si accorge che esiste un valore del "appn" diverso da meno uno, di default, allora non considera l'istanza (che in questo caso

sarebbe 14), ma il valore del “appn”, portando l’indice delo stack a quel valore, facendo sì che nell’esempio sia quattro e consideri l’istanza di quest’ultimo.

8. Modalità di compilazione

Come anticipato nel paragrafo 1, il file “*makefile.txt*” assolve proprio il compito di chiarire le modalità di compilazione ed inizia elencando quali sono i file necessari, tutti contenuti nella cartella “*progetto_tofu*”, incluso il “*makefile.txt*” stesso:

- “*def.h*”: file C che definisce i concetti base quali i nodi terminali e non, tramite enum, la union di valori interi e stringa, la struct del nodo e le funzioni ausiliarie di Lex, Yacc e tree.c senza implementazione.
- “*tree.c*”: include “*def.h*” e serve per stampare graficamente l’albero astratto generato dal programma tofu sul file "AbstractTree.txt".
- “*lexr.lex*”: include “*parser.h*” e “*def.h*”; è l’analizzatore lessicale.
- “*parser.y*”: include “*def.h*” e costituisce l’analizzatore sintattico.
- “semantics.c”: include “*parser.h*” e “*def.h*”; rappresenta l’analizzatore semantico e si occupa anche di stampare la Symbol Table del programma tofu sul file denominato appunto “SymbolTable.txt”.
- “*gen.c*”: include “semantics.c” e rappresenta il generatore di codice intermedio, il quale viene memorizzato sul file “IntermediateCode.txt”.
- “*interpreter.c*”: include “*gen.c*” ed è l’interprete/la macchina virtuale che traduce il T-code in istruzioni del linguaggio tofu.

Nota: il file di testo su cui scrive un file C se non esiste viene creato nuovo, altrimenti quello già esistente viene sovrascritto.
ATTENZIONE: i file sopra elencati, in generale, non devono essere modificati per alcun motivo; tuttavia, se ce ne fosse la necessità, è prerequisito inderogabile che l’utente in questione sia competente in materia!
Dopo queste informazioni preliminari in “*makefile.txt*” si trova una legenda così interpretabile:
output: file risultante/i, prodotto dall’attuale comando;
input: file richiesti per eseguire l’attuale comando;
comandi su linux: ciò che bisogna digitare su terminale linux, esattamente così come è riportato, affinché si abbia il risultato atteso.

Successivamente viene presentato l'elenco dei comandi, che sono i seguenti:

1) parser.h parser.c parser.output parser.dot: parser.y def.h
   bison -dvg -o parser.c parser.y

in cui l'opzione -d (header) genera il file parser.h che contiene le dichiarazioni delle informazioni esportabili (codifica dei simboli per Lex); l'opzione -v (verbose) produce parser.output che è la descrizione testuale della tabella di parsing LALR(1); infine l'opzione -g (graphic) genera parser.dot che rappresenta l'automa di parsing LALR(1) nel linguaggio dot.

2) parser.o parser.pdf: parser.c def.h parser.dot
   cc -g -c parser.c
   dot -Tpdf -o parser.pdf parser.dot

dove il primo comando genera parser.o , il file oggetto prodotto dal compilatore C (tutti i file con estensione .o), mentre il secondo comando produce il file parser.pdf, che raffigura a livello grafico il contenuto di parser.dot.

3) lexer.c: lexer.lex def.h
   flex -o lexer.c lexer.lex

I successivi tre comandi generano semplicemente file con estensione .o

4) lexer.o: lexer.c parser.h def.h
   cc -g -c lexer.c

5) tree.o: tree.c def.h
   cc -g -c tree.c

6) interpreter.o: def.h parser.h interpreter.c semantics.c gen.c
   cc -g -c interpreter.c

7) tofu_program: lexer.o parser.o tree.o interpreter.o
   cc -g -o tofu_program lexer.o parser.o tree.o interpreter.o

dove vengono passati in input tutti i file oggetto generati e si ottiene l'eseguibile denominato "tofu_program": ciò che serve realmente all'utente per far funzionare il programma scritto in tofu.
Dunque, l'ultimo e ottavo comando da eseguire su terminale linux verrà trattato nella prossima sezione, dato che l'utente deve scriverlo ogni qualvolta voglia eseguire un programma tofu, mentre i precedenti sette comandi è sufficiente che l'utente li esegua un'unica volta.

9. Uso per l'utente

Le regole lessicali e sintattiche che l'utente deve rispettare per scrivere un programma in linguaggio tofu (altamente consigliato scriverlo su un file di tipo .txt) sono specificate nei rispettivi analizzatori ("*lexr.lex*", "*parser.y*"), comunque le fondamentali sono le seguenti:

- Il corpo è una sezione obbligatoria e deve essere preceduta dalla key word "body", mentre le variabili e le funzioni sono sezioni opzionali, ma se presenti, bisogna farle precedere rispettivamente dalle key word "variables" e "functions";
- Un identificatore di variabili, funzioni o parametri deve iniziare con una lettera e può essere seguito in caso da una serie di caratteri solo alfanumerici;
- Ogni dichiarazione di variabili, di funzioni e statement del corpo deve essere seguito dal carattere ";" (punto e virgola);
- Se si vogliono dichiarare più variabili dello stesso tipo, lo si può fare in un'unica riga, separando col carattere ","(virgola) i rispettivi nomi;
- Una funzione può essere priva di parametri, ma deve obbligatoriamente ritornare un tipo (non esistono funzioni "void");
- Gli assegnamenti si possono fare solo all'interno del corpo, dunque le funzioni utilizzeranno solo parametri locali che verranno istanziati col valore delle variabili, che sono globali, quando la rispettiva funzione è invocata. È comunque possibile che una funzione al suo interno invochi un'altra funzione (che può coincidere con la stessa, quindi è ricorsiva);
- L'istruzione "show" di tofu, utilizzabile solo nel corpo, permette di visualizzare a video qualunque espressione sensata che la segue. Il carattere di ritorno a capo è definito come "\n". Se si vuole stampare il valore di un'espressione all'interno di una funzione, è necessario scrivere la seguente istruzione: "!(expr_stampata)";
- Infine, per richiedere un input dall'esterno si deve usare l'istruzione "?(tipo_valore_chiesto)".

Dopo aver seguito i passi indicati al paragrafo 8, l'utente dovrebbe avere a sua disposizione l'eseguibile "tofu_program", grazie al quale il programma tofu passato in input può essere compilato digitando l'ultimo comando linux richiesto:

8) ./tofu_program (eventuali input richiesti all'utente) < program.txt

dove "program.txt" è il programma tofu scritto dall'utente. Invece la scritta fra parentesi tonde indica che l'utente in questo comando deve inserire in ordine gli

input richiesti dal "program.txt" separati da uno spazio. Per esempio, se sono presenti "?(int)", "?(string)", "?([bool])" in questa sequenza, allora il comando indicato sopra si può scrivere così:

./tofu_program 12 ciao [true,false] < program.txt

Se tuttavia non è richiesto alcun input esterno, allora il comando sarà semplicemente: ./tofu_program < program.txt

Per inserire una lista, l'utente deve seguire le seguenti indicazioni:

i. Inserire il numero giusto di parentesi quadre, cioè non ci deve essere una parentesi "[" o "]" pendente; inoltre, la lista inserita deve essere coerente con il livello di profondità richiesto;
ii. Non lasciare spazi fra le quadre e gli elementi, separare quest'ultimi tramite l'uso della "," (virgola), senza digitare nessun carattere di spaziatura;
iii. Gli elementi inseriti possono essere solo costanti semplici, quindi che non siano risultato di alcuna operazione. Ad esempio, in una lista di interi, "30" è accettato, ma "10+20" non viene riconosciuto.

Se si inserisce una lista non rispettando una delle seguenti regole, il programma potrebbe avere dei comportamenti anomali o terminare con un errore, in quanto non è in grado di gestire i casi sopra elencati. Invece è in grado di effettuare dei controlli fra il tipo da inserire dichiarato e l'input effettivamente fornito.
L'output del comando risulta essere i file citati nel paragrafo 8, quindi "AbstractTree.txt", "SymbolTable.txt" e "IntermediateCode.txt", la visualizzazione su terminale linux dell'assenza di errori in ogni funzione e statement, che viene generata dall'analizzatore semantico, ed infine la stampa a video di ciò che l'utente ha richiesto in tofu tramite la "show" o la "!(expr)"; nel secondo caso, dopo la stampa, è impostato di default un ritorno a capo.
Nella cartella "*progetto_tofu*" è presente un programma d'esempio denominato "*tofu_example.txt*", per cui non sono richiesti input esterni, che l'utente è libero di provare a compilare.