



*DIPARTIMENTO DI
INGEGNERIA DELL'INFORMAZIONE*

Corso di Laurea
in Ingegneria Informatica

Relazione Finale

**Progettazione e sviluppo di un linguaggio
di programmazione ibrido**

Relatore: Prof. Gian Franco Lamperti

Laureando:
Dario Filippini Toninato
Matricola n. 727825

Anno Accademico 2021/2022

Indice

| | |
|---|-----------|
| 1 Introduzione | 4 |
| 2 Linguaggi di programmazione | 5 |
| 2.1 Classificazione dei linguaggi | 6 |
| 2.1.1 Linguaggi imperativi | 6 |
| 2.1.2 Linguaggi orientati agli oggetti | 7 |
| 2.1.3 Linguaggi funzionali | 9 |
| 2.1.4 Linguaggi logici | 10 |
| 2.1.5 Linguaggi ibridi | 11 |
| 2.2 Specifica di un linguaggio | 11 |
| 2.2.1 Lessico | 12 |
| 2.2.2 Sintassi | 13 |
| 2.2.3 Semantica | 14 |
| 3 Progettazione di un linguaggio di programmazione | 17 |
| 3.1 Tipologie di un linguaggio | 18 |
| 3.1.1 Linguaggio compilato | 18 |
| 3.1.2 Linguaggio interpretato | 21 |

| | |
|---|------------|
| 3.1.3 Linguaggio pseudocompilato | 22 |
| 3.2 Analisi di un linguaggio | 23 |
| 3.2.1 Analisi lessicale | 24 |
| 3.2.2 Analisi sintattica | 26 |
| 3.2.3 Analisi semantica | 32 |
| 4 Il linguaggio di programmazione Tofu | 35 |
| 5 Il traduttore di Tofu | 43 |
| 5.1 Il lexer | 45 |
| 5.2 Il parser | 47 |
| 5.3 L'analizzatore semantico | 55 |
| 5.4 Il codice intermedio T-code | 66 |
| 5.4.1 Gli schemi di traduzione | 66 |
| 5.4.2 Il generatore di T-code | 72 |
| 6 La macchina virtuale di Tofu | 79 |
| 7 Installazione del software | 89 |
| 8 Conclusioni | 93 |
| Bibliografia e Sitografia | 94 |
| Appendice A | 96 |
| Appendice B | 100 |

| | |
|-----------------------|------------|
| Appendice C | 108 |
| Appendice D | 123 |
| Appendice E | 136 |
| Ringraziamenti | 152 |

1 Introduzione

Il percorso formativo, sia accademico che personale, di un ingegnere informatico è improntato verso l’acquisizione di un’ampia e approfondita conoscenza dei linguaggi di programmazione. Infatti sono uno strumento potente e malleabile, grazie ai quali si possono sviluppare software delle più svariate tipologie. Ad un certo punto, semplicemente per curiosità o per la necessità di comprendere come funziona un particolare linguaggio nei minimi dettagli, ci si potrebbe domandare come viene progettato e sviluppato un linguaggio di programmazione. La seguente relazione risponde proprio a questa domanda: sarà presentato un nuovo linguaggio di nome **Tofu** e verranno spiegati tutti i passaggi seguiti per la sua realizzazione. Come per un linguaggio naturale, anche per uno artificiale bisogna definirne il lessico e la sintassi; su questa struttura basilare si appoggia la semantica, il cui compito è dare un senso compiuto alle frasi. Ciò rientra nella fase di ideazione, in cui inoltre si decide la classificazione (imperativo, ad oggetti, funzionale o logico), le operazioni (aritmetiche, logiche, ecc...), le strutture dati ed eventuali altre specifiche del linguaggio. L’ulteriore passaggio consiste nell’implementare l’analizzatore lessicale, sintattico e semantico, basati sulle scelte fatte in fase di progettazione e il cui funzionamento verrà sviscerato nel corso dei capitoli. Da questo punto si possono prendere diverse strade a seconda che si decida di realizzare un linguaggio di tipo compilato, interpretato o pseudocompilato. **Tofu** ricade nell’ultima casistica ed è un ibrido fra il paradigma imperativo e funzionale, perciò si vedranno nel dettaglio tali aspetti, ma molti concetti qui espressi potrebbero essere utilizzati per creare un linguaggio di programmazione con caratteristiche differenti. Per comprendere meglio gli aspetti trattati, si faranno esempi di codice, di programmi **Tofu** e della loro esecuzione.

2 Linguaggi di programmazione

Un linguaggio di programmazione è una notazione formale che specifica un insieme di istruzioni che possono essere usate per produrre dati in uscita; esso può essere definito anche come uno strumento di astrazione che permette di specificare computazioni tali da poter essere eseguite su un elaboratore. La ragion d'essere dei linguaggi di programmazione è rendere più facile l'uso delle macchine fisiche, tramite la componente software, che è fondamentale tanto quanto quella hardware. Si può considerare l'hardware come il corpo umano, mentre il software come la mente umana: nozioni che da sole non possono esistere e che quindi hanno intrinsecamente bisogno l'una dell'altra. L'idea di software può spaziare dal concetto di sistema operativo, a quello di programma stesso, fino ad arrivare nell'ambito delle applicazioni che vengono usate quotidianamente dagli utenti. Dunque, nell'era moderna il significato di macchina fisica è più sfumato: si possono intendere i classici dispositivi elettronici come computer, ipad e smartphone, oppure qualsiasi altro dispositivo dotato di software come ad esempio la maggior parte degli elettrodomestici o delle automobili.

Storicamente, intorno alla fine degli anni '40 e l'inizio degli anni '50 fu introdotto uno dei primi linguaggi: l'Assembly, che costituisce una rappresentazione simbolica, cioè mnemonica, del linguaggio macchina. In pratica l'Assembly viene tradotto automaticamente in linguaggio macchina ed è strettamente legato al modo in cui funziona il calcolatore più che con l'oggetto della computazione; infatti le sue istruzioni corrispondono a quelle che il processore può eseguire. Per tale ragione viene considerato un linguaggio con un basso livello di astrazione, mentre ciò che si desiderava era lo sviluppo di linguaggi general-purpose, quindi versatili e con un ampio dominio applicativo, e di alto livello, cioè indipendenti dalla macchina. Ciò avrebbe

permesso di soddisfare la proprietà di portabilità che consente di scrivere programmi in grado di funzionare correttamente su piattaforme hardware diverse e sotto differenti sistemi operativi, richiedendo semplicemente la ricompilazione dei sorgenti nel nuovo ambiente. I primi linguaggi di alto livello furono il FORTRAN (1957) e l'ALGOL (1960), che può essere considerato il capostipite dei linguaggi moderni. Nel corso dei decenni sono stati sviluppati migliaia di linguaggi programmazione, ognuno progettato in modo da soddisfare certi requisiti e per determinati usi; oggi alcuni tra i più diffusi in assoluto sono Python, Java, JavaScript, C++, C ed R.

2.1 Classificazione dei linguaggi

I linguaggi vengono distinti tipicamente in quattro grandi famiglie basate sul paradigma di programmazione di riferimento, ovvero lo stile di programmazione: i linguaggi imperativi, quelli orientati agli oggetti, quelli funzionali e quelli logici.

2.1.1 Linguaggi imperativi

Il programma è costituito da una sequenza di passi o istruzioni, ciascuna delle quali può essere pensata come un “ordine” che viene impartito alla macchina virtuale del linguaggio di programmazione utilizzato. Da un punto di vista sintattico, i costrutti sono spesso identificati da verbi all'imperativo (da cui deriva il nome del paradigma). Ogni istruzione è un comando esplicito, che opera su una o più variabili oppure sullo stato interno della macchina, e le istruzioni vengono eseguite in un ordine prestabilito. Quindi ad ogni passo avviene una lettura dell'input, una computazione oppure una scrittura dell'output. I costrutti fondamentali sono gli assegnamenti, le sequenze, le istruzioni condizionali (if-else, switch), i cicli (for, while, do-while) e la loro interruzione (break); inoltre il calcolo procede per iterazione piuttosto che per ricorsione. I valori delle variabili sono spesso assegnati a partire da costanti o da

altre variabili e raramente per passaggio di parametri (istanziazione).

Esempi: FORTRAN, Cobol, Pascal, C.

```
#include <stdio.h>
int main() {
    int number1, number2, sum;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);
    // calculating sum
    sum = number1 + number2;
    printf("%d + %d = %d", number1, number2, sum);
    return 0;
}
```

Figura 2.1: Esempio di programma C per calcolare la somma di due interi

2.1.2 Linguaggi orientati agli oggetti

Il programma è una collezione di oggetti che interagiscono gli uni con gli altri scambiandosi messaggi che trasformano il loro stato. La programmazione ad oggetti fornisce un supporto naturale alla modellazione software degli oggetti del mondo reale o del modello astratto da riprodurre ed è particolarmente adatta quando esistono delle relazioni di interdipendenza tra quest'ultimi.

Tale paradigma prevede di raggruppare in alcune parti circoscritte del codice sorgente, chiamate classi, la dichiarazione delle strutture dati e delle procedure che operano su di esse. Le classi, quindi, costituiscono dei modelli astratti, che a tempo di esecuzione vengono invocate per istanziare o creare oggetti software relativi alla classe invocata. Questi ultimi sono dotati di attributi (dati) e metodi (procedure) secondo quanto definito dalle rispettive classi. La parte del programma che fa uso di un oggetto si chiama client. I costrutti fondamentali sono l'incapsulamento, l'ereditarietà e il polimorfismo.

- *L'incapsulamento* è la proprietà per cui i dati che definiscono lo stato interno

di un oggetto e i metodi che ne definiscono la logica sono accessibili ai metodi dell’oggetto stesso, mentre non sono visibili ai client e dunque tali metodi si dicono privati. Per alterare lo stato interno dell’oggetto è necessario invocarne i metodi che vengono definiti pubblici, cioè accessibili dall’esterno.

- *L’ereditarietà* permette di derivare nuove classi a partire da quelle già definite, realizzando in tale maniera una gerarchia di classi. Una classe derivata attraverso l’ereditarietà (sottoclasse o classe figlia) mantiene i metodi e gli attributi della classe da cui deriva (superclasse o classe madre); inoltre può definire i propri metodi o attributi e ridefinire il codice di alcuni dei metodi ereditati tramite un meccanismo chiamato overriding. Una classe può avere più sottoclassi, ma una sola superclasse.
- *Il polimorfismo* permette di specificare un client che può servirsi di oggetti di classi diverse, ma dotati di una stessa interfaccia comune; nel tempo di esecuzione tale client potrà attivare comportamenti diversi senza conoscere a priori il tipo specifico dell’oggetto che gli viene dato in ingresso. Ciò è possibile grazie alla struttura dell’interfaccia, priva di attributi ed in cui è permesso dichiarare solo metodi astratti, in modo che possano avere comportamenti diversi per ogni classe che implementa quell’interfaccia.

Esempi: Java, Python, Smalltalk, C#, Ruby.

```

interface Backend {
    // abstract class
    public void connectServer();
}

class Frontend {

    public void responsive(String str) {
        System.out.println(str + " can also be used as frontend.");
    }
}

// Language extends Frontend class
// Language implements Backend interface
class Language extends Frontend implements Backend {

    String language = "Java";

    // implement method of interface
    public void connectServer() {
        System.out.println(language + " can be used as backend language.");
    }

    public static void main(String[] args) {
        // create object of Language class
        Language java = new Language();
    }
}

```

Figura 2.2: Esempio di programma Java che implementa l'ereditarietà multipla

2.1.3 Linguaggi funzionali

Il programma è una collezione di funzioni matematiche, con un rispettivo dominio e codominio, che vengono valutate durante il flusso di esecuzione. Il punto di forza principale di questo paradigma è la mancanza di effetti collaterali (side-effects) delle funzioni, il che comporta una più facile verifica della correttezza e della mancanza di bug del programma e la possibilità di una maggiore ottimizzazione dello stesso. Si dice che una funzione produce un effetto collaterale quando modifica un valore o uno stato al di fuori del proprio scoping locale, modificando ad esempio una variabile globale/statica o uno dei suoi argomenti. In un linguaggio funzionale puro non esistono le variabili, le istruzioni di controllo e risulta assente l'assegnazione, dato che si utilizza soltanto il passaggio dei parametri. Tipicamente in tale modello il controllo del calcolo è gestito dalla ricorsione e dal pattern matching, cioè l'azione di controllo della presenza di un certo motivo - pattern - all'interno di un oggetto

composito; la struttura dati più diffusa è la lista, una sequenza di elementi.

Esempi: Lisp, Scheme, Haskell.

```
(define (reverse A)
  (if (null? A) ()
      (append (reverse (cdr A)) (list (car A))))
  ))
```

Figura 2.3: Esempio di programma Scheme che computa l'inverso di una lista atomica

2.1.4 Linguaggi logici

Il programma consiste in un insieme di dichiarazioni logiche (o “affermazioni” o clausole, non “ordini”) che la macchina virtuale del linguaggio è (implicitamente) tenuta a considerare vere e/o rendere vere. Ci si concentra più sulla descrizione della struttura logica del problema e il miglior modo di rappresentarla piuttosto che sul modo di risolvere il problema e pervenire ai risultati. Durante l'esecuzione di un programma si applicano le varie clausole per identificare possibili soluzioni al problema, cercando di trovare l'ordine giusto in cui eseguire le dichiarazioni, tipicamente procedendo per “tentativi”. Il calcolo procede per ricorsione invece che per iterazione e per uno stesso problema si può giungere a soluzioni distinte: si applica il nondeterminismo. Inoltre una proprietà intrinseca del linguaggio è il backtracking, cioè la possibilità di tornare sui propri passi per percorrere una strada alternativa a quella seguita in precedenza.

Esempio: Prolog.

```
fratello(X, Y) :- genitore(Z, X), genitore(Z, Y), X \= Y.  
cugino(X, Y) :- genitore(Z, X), genitore(W, Y), fratello(Z, W).
```

Figura 2.4: Esempio di programma Prolog che, basandosi sulla clausola genitore, specifica le relazioni di fratello e cugino

2.1.5 Linguaggi ibridi

Esistono linguaggi che sono orientati verso due o più paradigmi di programmazione, permettendo una maggiore libertà a scapito di una minore chiarezza, e che vengono chiamati linguaggi ibridi. Per esempio, il linguaggio general-purpose C++ appartiene a questa categoria, essendo una versione orientata ad oggetti del C: soddisfa sia il paradigma imperativo che quello ad oggetti. Il C++ è compatibile con il C, nel senso che il codice C può essere compilato da un compilatore per C++ e quindi il codice C esistente può essere inserito nei programmi scritti in C++. In questo linguaggio ibrido esistono alcuni tipi di dati che non vengono trattati come oggetti, ad esempio i tipi int, float e char, mentre nei linguaggi puri ad oggetti ogni cosa è vista come un oggetto.

Il nuovo linguaggio che verrà presentato in seguito con il nome di Tofu è ibrido: combina il paradigma imperativo con quello funzionale.

2.2 Specifica di un linguaggio

Il livello di formalizzazione di un linguaggio di programmazione, detto anche linguaggio artificiale, idealmente non dovrebbe essere né troppo rigoroso, per evitare di creare un problema di accettazione del linguaggio da parte dei destinatari, né troppo informale, affinché non possano esserci ambiguità e una proliferazioni di varianti del linguaggio. La sua descrizione, come per i linguaggi naturali, comprende di specificare il lessico, la sintassi e la semantica delle sue frasi; inoltre deve essere sia concisa che comprensibile.

2.2.1 Lessico

Per definizione, come visto all'inizio di questo capitolo, un linguaggio di programmazione è una notazione formale: un insieme di stringhe costruite sopra un determinato alfabeto. Un alfabeto è un insieme finito di caratteri, mentre una stringa è una sequenza finita e significativa di caratteri dell'alfabeto.

Esempio di alfabeto: $\{0,1\}$ = alfabeto binario.

Esempio di stringa: $\{A,\dots,Z,a,\dots,z\}\{0,\dots,9\}$ = stringhe composte da una lettera seguita da una cifra (C3, dx7).

Il lessico si può considerare parte integrante della sintassi, dato che esprime regole strutturali, tuttavia meglio specificarlo a parte in quanto descrive le unità sintattiche di più basso livello. Per tale ragione, ha una complessità piccola che gli permette di possedere una standardizzazione, la quale viene fornita dalle espressioni regolari: un potente formalismo usato per specificare un pattern, ovvero una regola per descrivere le istanze di un simbolo (astrazione di una classe di stringhe lessicali).

Esempio di espressione regolare:

Simbolo: id = identificatore.

Istanze: x, num12, contatore, T101.

Pattern: identificatore che inizia obbligatoriamente con una lettera e che può essere opzionalmente seguito da lettere e/o cifre.

Espresione regolare: $id = \text{lettera} (\text{lettera} \mid \text{cifra})^*$.

I linguaggi che possono essere determinati da un'espressione regolare vengono definiti *insiemi regolari* e tutti quelli artificiali comunemente usati soddisfano questo requisito. Per approfondire le espressioni regolari consultare l'Appendice A.

2.2.2 Sintassi

Le regole sintattiche specificano la struttura delle frasi di un linguaggio, definite come stringhe di caratteri su un certo alfabeto. La sintassi svolge un ruolo con una duplice valenza: riconoscitore e generatore di un linguaggio. Il riconoscitore ha il compito di determinare se una frase appartiene o meno al linguaggio in questione, ma ha poca utilità perché è uno strumento che viene usato per tentativi. Il generatore ha la responsabilità di generare delle frasi appartenenti al linguaggio, però risulta uno strumento imprevedibile. Allora si è deciso di sfruttare un meccanismo del riconoscitore basato su quello del generatore; il riconoscitore stesso prova a generare la frase che sta analizzando: se ci riesce significa che questa appartiene al linguaggio, altrimenti no. La sintassi presenta una complessità media e dunque è ancora possibile standardizzarla mediante due formalismi quasi equivalenti: lo strumento 2 di Chomsky¹, le grammatiche non contestuali, e la BNF (Backus-Naur Form)². La BNF viene considerata un metalinguaggio e l'idea di fondo della sua notazione è usare astrazioni per definire le strutture sintattiche, di solito in maniera ricorsiva. L'astrazione è un simbolo nonterminale, mentre la stringa lessicale è un simbolo terminale: questi due elementi compongono le produzioni; la grammatica BNF è definita come un insieme di produzioni. Partendo dall'assioma, cioè l'astrazione di più alto livello, si genera passo dopo passo, mediante una serie di applicazioni delle regole, una frase del linguaggio: questo è il processo di derivazione.

¹Avram Noam Chomsky (nato nel 1928) è filosofo, linguista, accademico e scienziato statunitense

²John Warner Backus (1924-2007) è stato un informatico e matematico statunitense

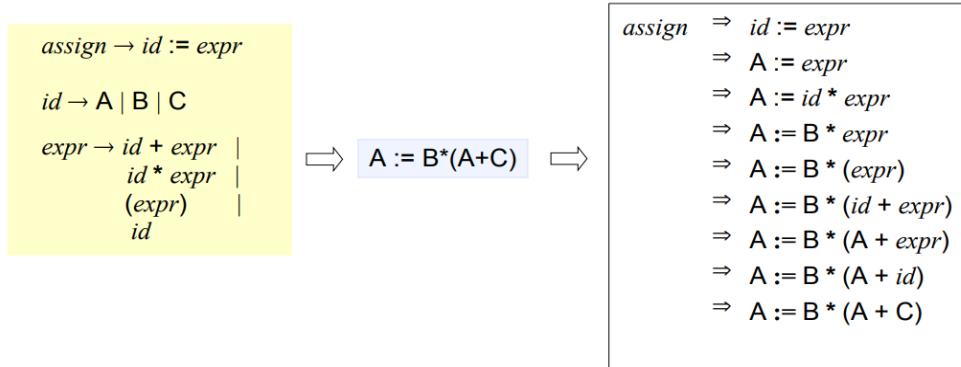


Figura 2.5: Esempio di grammatica BNF a sinistra, di frase concreta del linguaggio in centro, di derivazione a destra

Nella BNF della figura 2.5:

- $assign \rightarrow id := expr$ è l'assioma;
- $id \rightarrow \dots$ e $expr \rightarrow \dots$ sono le produzioni;
- $assign, id, expr$ sono i simboli nonterminali;
- $:=, A, B, C, +, *, ()$ sono i simboli terminali.

La struttura gerarchica delle frasi può essere rappresentata anche tramite alberi sintattici. Oltre alla grammatica BNF, esiste anche la EBNF (Extended BNF), dotata dello stesso potere espressivo della BNF, ma che introduce nuovi metacaratteri, aumentando sia la leggibilità che la scrivibilità. Per ulteriori dettagli sugli argomenti qui trattati, ci si riferisca all'Appendice B.

2.2.3 Semantica

Specifica il significato delle frasi di un linguaggio in fase di esecuzione, cioè dinamicamente. È strettamente correlata alla sintassi: la sintassi deve far intuire la semantica (il termine *if* suggerisce un costrutto condizionale) ed allo stesso tempo la semantica deve essere guidata dalla sintassi. La complessità risulta grande, perciò non esiste

una standardizzazione; i formalismi comunemente impiegati sono due: semantica operazionale e denotazionale.

La semantica operazionale descrive l'esecuzione di un programma attraverso transizioni definite direttamente sul linguaggio del programma ed è espressa in termini algoritmici. È concettualmente simile all'interpretazione vera e propria in cui abbiamo una macchina virtuale e le istruzioni applicano transizioni di stato in questa macchina.

| <i>Operatore relazionale</i> | <i>Semantica operazionale</i> |
|----------------------------------|---|
| <code>S := select [p] R</code> | <pre> S := {}; n := length(R); for i := 1 to n do begin t := R[i]; ok := p(t); if ok = true then insert(t, S) end. </pre> |

Figura 2.6: Esempio di semantica operazionale dell'operatore di selezione sull'insieme R

La semantica denotazionale è un formalismo più rigoroso che descrive il significato di un programma, a prescindere dalla sua esecuzione, e si basa sulla teoria delle funzioni ricorsive. L'idea è quella di denotare matematicamente, con un dominio, ogni astrazione sintattica del linguaggio; le istanze dell'astrazione corrispondono agli elementi del dominio.

Come esempio proviamo ad esprimere la semantica dei numeri decimali, basandoci sulla BNF:

$$dnum \rightarrow 0 \mid 1 \mid \dots \mid 9 \mid dnum\ 0 \mid dnum\ 1 \mid \dots \mid dnum\ 9$$

$$\begin{aligned}
 M_d('0') &= 0, M_d('1') = 1, \dots, M_d('9') = 9 \\
 M_d(dnum '0') &= 10 * M_d(dnum) \\
 M_d(dnum '1') &= 10 * M_d(dnum) + 1 \\
 &\dots \\
 M_d(dnum '9') &= 10 * M_d(dnum) + 9
 \end{aligned}$$

Figura 2.7: Semantica denotazionale dei numeri decimali

Facciamo degli esempi per chiarire il significato della figura 2.7.

Il numero 8 viene semplicemente valutato con il valore di 8.

Il valore del numero 65 viene calcolato nel seguente modo:

$$M_d(dnum '5') = 10 * M_d('6') + 5 = 10 * 6 + 5 = 60 + 5$$

3 Progettazione di un linguaggio di programmazione

Una buona progettazione di un linguaggio è fondamentale dato che è lo strumento impiegato per lo sviluppo di qualsiasi tipologia di software: un linguaggio con un'alta qualità produrrà software di ottima qualità e viceversa. Un indicatore del livello di qualità del software è rappresentato dalla sua affidabilità, cioè l'essere in grado di rispettare i suoi requisiti in ogni circostanza. L'affidabilità è supportata dalle seguenti caratteristiche di un linguaggio:

- *Una buona scrivibilità*: misura di quanto facilmente può essere usato per scrivere un programma relativo ad un certo dominio applicativo (scientifico, gestionale). I linguaggi di basso livello come l'Assembly sono meno scrivibili di quelli di alto livello.
- *Una buona leggibilità*: misura della possibilità di seguire la logica del programma leggendolo. Anche questa caratteristica risulta migliore nei linguaggi di alto livello rispetto che in quelli di basso livello.
- *Semplicità*: grado di riduzione del numero di costrutti. Infatti un linguaggio con molti costrutti base risulta essere complicato da imparare.
- *Sicurezza*: mancanza o minimizzazione di costrutti che permettono la scrittura di programmi pericolosi. Due esempi sono l'istruzione goto e i puntatori. Il goto salta in maniera incondizionata a qualsiasi altra istruzione del programma, quindi amplia l'elenco delle istruzioni successive. I puntatori espandono l'insieme delle celle di memoria referenziate da una variabile. In entrambe le

casistiche, il pericolo consiste nel fatto che un errore può manifestarsi in una parte di codice lontana dal punto che ne è in realtà la causa.

- Robustezza: grado di capacità di reazione ad eventi indesiderati, quali overflow aritmetico, esaurimento memoria ed input errato. La possibilità di intrappolare tali eventi e definire una risposta appropriata alla loro manifestazione è ciò che si chiama gestione delle eccezioni.

3.1 Tipologie di un linguaggio

Prerequisito indispensabile prima di poter progettare un linguaggio è conoscere la differenza tra linguaggi compilati, interpretati e pseudocompilati. Questo perché bisogna decidere se scrivere un compilatore o un traduttore per il linguaggio quando lo si implementa; tuttavia si tenga presente che i concetti di interpretazione e compilazione non sono afferenti al linguaggio in sè: ogni linguaggio può essere implementato in entrambi i modi. Per comprendere la distinzione tra le varie implementazioni sarà necessario definire con precisione cosa si intende quando si parla di compilatore, traduttore e interprete.

3.1.1 Linguaggio compilato

Il codice sorgente di un linguaggio compilato viene tradotto in codice macchina dal compilatore ed eseguito direttamente dal processore, senza l'ausilio di nessun tipo di software di traduzione intermedio. Inizialmente il programma sorgente scheletrico, viene passato al preprocessore che si occupa di operazioni come l'inclusione di altri file o l'espansione delle macro, cioè un blocco di comandi o istruzioni tipicamente ricorrente, producendo in output il programma sorgente vero e proprio. Questo viene poi dato in input al compilatore, che svolge la maggior parte del lavoro, e restituisce il programma target Assembly, che è tradotto in codice macchina rilocabile

dall’assemblatore. Poi il linker ha il compito di collegare i diversi file oggetto con le librerie usate all’interno del programma e di combinarli in un unico file eseguibile. Questo viene caricato in memoria, insieme alle librerie necessarie, dalla componente del sistema operativo denominata loader. L’intero processo finora descritto è la fase del ciclo di vita del programma che avviene a tempo di compilazione (compile-time). Infine il file eseguibile costituito dal codice macchina assoluto può essere eseguito dalla CPU: questa fase del ciclo di vita del programma avviene a tempo di esecuzione (runtime).

Il compilatore è a sua volta un programma che è strutturato in due macromoduli: il primo, in cui avviene una procedura di analisi, è il *front-end* e il secondo, nel quale si attua una sintesi, è il *back-end*. L’analisi consiste nel riconoscere le parole del programma e mappare le sue operazioni su una struttura gerarchica: l’albero sintattico; la sintesi si occupa della costruzione del programma target Assembly: è un processo più complesso e specializzato dell’analisi. Nello stadio di front-end si traduce il codice sorgente in un linguaggio intermedio, che di solito è interno al compilatore; nello stadio di back-end, partendo dal codice intermedio, avviene la generazione del codice target. Le fasi del front-end comprendono l’analisi lessicale, quella sintattica, quella semantica (approfondite nella sezione 3.2) e la generazione di codice intermedio. Le fasi del back-end sono l’ottimizzazione del codice intermedio, la generazione del codice target e la sua ottimizzazione. L’organizzazione in moduli conferisce più flessibilità al compilatore: bisogna modificare il front-end se varia il linguaggio di programmazione, tuttavia non se cambia l’hardware sottostante; viceversa per il back-end. Si è scelto di operare due ottimizzazioni in base a quanto appena detto, ma anche perché il modo di ottimizzare il codice è mutevole in base al contesto. Inoltre il linguaggio intermedio permette di facilitare il lavoro del compilatore.

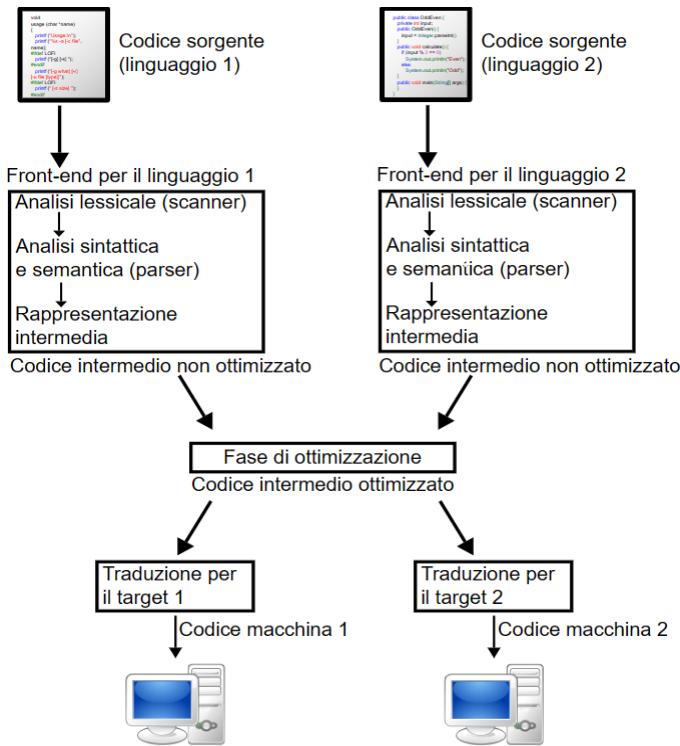


Figura 3.1: Schema di un compilatore

Il principale vantaggio di questo linguaggio è la sua velocità nella fase di run (esecuzione), che è ottenuta con la creazione di file eseguibili con minor “peso” in termini di tempo e memoria, essendo in grado di adattare vari parametri di questa fase all’hardware a disposizione. Tuttavia, proprio in virtù di quanto appena affermato, il suo maggior svantaggio è quello di essere vincolato a una piattaforma (combinazione fra architettura hardware e sistema operativo) particolare. Infatti è necessario compilare un eseguibile diverso per ogni piattaforma: la portabilità è limitata. Dunque se si sceglie di progettare un linguaggio compilato si deve conoscere approfonditamente la piattaforma sul quale girerà e la tipologia di linguaggio Assembly destinato ad essa. Oggi la piattaforma più diffusa è la 80x86 a 32/64 bit accompagnata dal sistema operativo Windows, che fa uso dell’Assembly 80x86.

Alcuni esempi di linguaggi compilati sono C, C++ e Haskell.

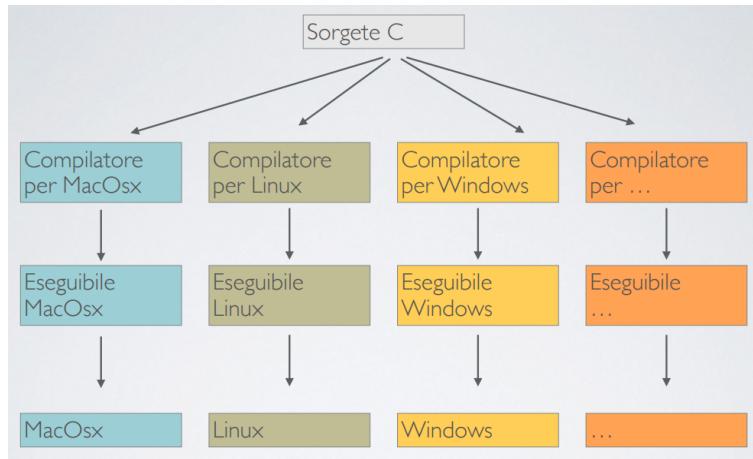


Figura 3.2: Schema di compilazione del C

3.1.2 Linguaggio interpretato

Il codice sorgente di un linguaggio interpretato viene passato all’interprete, un programma che lo esegue direttamente. Un interprete è composto da un analizzatore lessicale, sintattico, semantico e da un programma che, basandosi sull’albero sintattico, deve prima identificare le azioni da eseguire e poi eseguirle, effettuando l’esecuzione vera e propria. Bisogna far presente che l’interprete per essere in grado si svolgere il suo compito, a differenza del compilatore, introduce dell’overhead: risorse in più e accessorie rispetto a quelle strettamente richieste. Ciò comporta una minore efficienza a run-time, dato che il programma richiede maggior quantità di tempo e memoria. Inizialmente i linguaggi interpretati erano molto più lenti di quelli compilati, ma si è scelto di continuare a svilupparli a causa del loro principale vantaggio: l’elevata portabilità. Infatti poiché l’interprete ha al suo interno delle librerie compilate ad hoc per ogni piattaforma, può eseguire direttamente il codice sorgente, senza la necessità di ricompilarlo per ciascuna piattaforma, a differenza dei linguaggi compilati. Per risolvere il problema dell’inefficienza a run-time si è pensato di introdurre approcci ibridi.

Alcuni esempi di linguaggi interpretati sono Perl, PHP e JavaScript.

3.1.3 Linguaggio pseudocompilato

Il linguaggio pseudocompilato è una soluzione ibrida tra il linguaggio compilato e quello interpretato, pensata per avere i principali vantaggi dei due approcci. Il codice sorgente è dato in input ad un traduttore/compilatore che però non produce un codice macchina, bensì un codice intermedio (diverso dal programma target Assembly generato dal compilatore tradizionale). Quest'ultimo in seguito può venire sia interpretato sia compilato al momento dell'esecuzione: se si sceglie la prima opzione si parla di *macchina virtuale* (virtual machine), altrimenti si parla di *compilatore just-in-time* (lett. "appena in tempo").

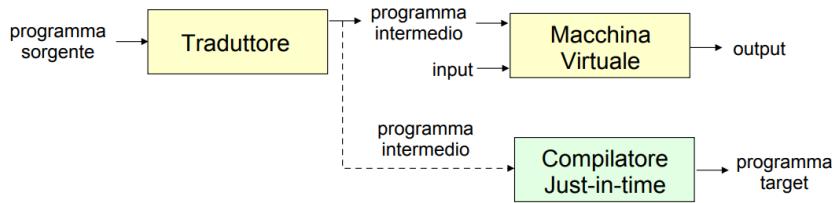


Figura 3.3: Schema di un linguaggio pseudocompilato

Il traduttore si occupa dell'analisi lessicale, sintattica e semantica del programma di alto livello, dopodiché, basandosi sull'albero sintattico, deve tradurre le sue istruzioni in quelle del linguaggio intermedio, che può avere un livello di astrazione più o meno elevato. Il primo linguaggio pseudocompilato è stato il Pascal con l'introduzione del P-code: un codice intermedio di più alto livello rispetto all'Assembly, riuscendo a descrivere con un solo comando operazioni moderatamente complesse come la stampa a video o il caricamento di una variabile. In seguito furono sviluppati linguaggi come Java e Python che sfruttano il bytecode, un linguaggio intermedio molto simile all'Assembly e chiamato così perché spesso le operazioni hanno un codice che occupa un solo byte.

La macchina virtuale è l'interprete del programma intermedio (P-code, bytecode o di altro genere) e genera l'output del programma sorgente; invece il compilatore just-in-time effettua una traduzione dinamica durante l'esecuzione del codice ricevuto in input, cioè compila le sue istruzioni appena prima della loro esecuzione (da cui il nome), e produce in uscita il codice macchina.

Grazie al suo funzionamento, questa soluzione ibrida risulta notevolmente più veloce a tempo di esecuzione rispetto a un linguaggio interpretato, seppur rimanendo più lenta di uno compilato; inoltre mantiene un'ottima portabilità, molto superiore a quella del linguaggio compilato, anche se inferiore a quella del linguaggio interpretato, dato che la macchina virtuale/il compilatore just-in-time devono essere adattati a una specifica piattaforma.

Il linguaggio Tofu appartiene a questa tipologia.

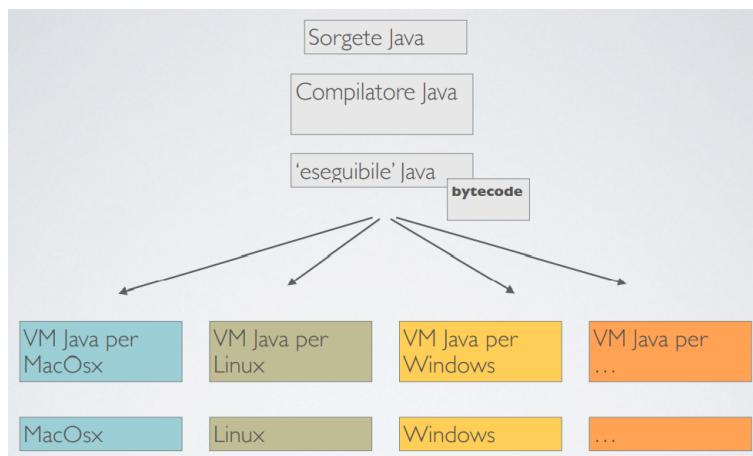


Figura 3.4: Schema di pseudocompilazione di Java

3.2 Analisi di un linguaggio

A prescindere dall'implementazione scelta, è necessario sottoporre il linguaggio a una fase di analisi; ne esistono tre tipologie: quella lessicale, sintattica e semantica. Il procedimento di analisi, come si può intuire, è strettamente connesso alla specifica di

un linguaggio (capitolo 2.2) e viene incorporato sia dal compilatore che dall’interprete che dal traduttore, come visto nel capitolo precedente. Un suo compito essenziale è quello di gestire i possibili errori: man mano che l’analisi procede il sorgente viene passato “al setaccio” e “filtrato”. Se è tutto corretto, si passa allo step successivo, altrimenti il programma deve terminare e segnalare un messaggio d’errore all’utente.

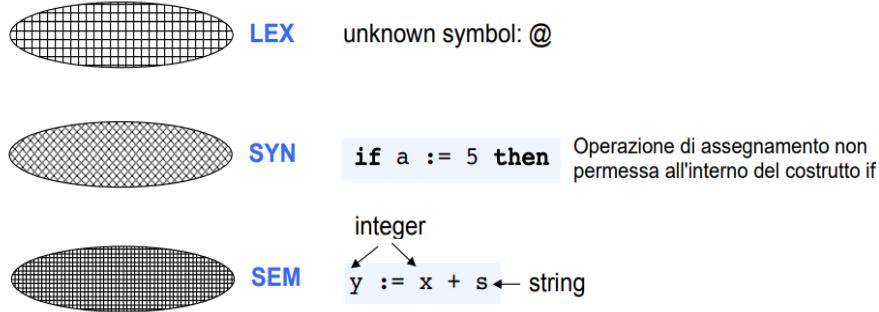


Figura 3.5: Schema di gestione degli errori durante il processo di analisi

3.2.1 Analisi lessicale

Il ruolo primario dell’analisi lessicale è definire quali raggruppamenti di caratteri, detti stringhe lessicali, possano essere trasformati, mediante un processo di astrazione, in simboli (token). Quando un simbolo rappresenta molteplici stringhe lessicali viene denominato attributo lessicale. Il principale strumento di riferimento per questo tipo di analisi sono le espressioni regolari (argomento trattato nel capitolo 2.2 e approfondito nell’Appendice A), che permettono di definire la regola (pattern) che collega un simbolo con la sua stringa lessicale.

Esempi:

- $eq \rightarrow ==$

dove eq , abbreviazione di equal, è un simbolo e $==$ una stringa lessicale che indica l’operazione di uguaglianza.

- $boolconst \rightarrow true \mid false$

dove $boolconst$, che sta per costante booleana, è un attributo lessicale, mentre $true$ e $false$ sono due stringhe lessicali.

Il ruolo secondario dell’analizzatore lessicale consiste nel rimuovere la spaziatura e i commenti dal sorgente e di contare le linee di codice del programma, in modo di essere in grado di segnalare all’utente il numero di linea in cui si è verificato un errore. Inoltre deve tener traccia del reale valore di un simbolo all’interno del programma, cioè delle coppie token-valore: ad esempio $(id, var1)$ o $(boolconst, true)$. I token sono caratterizzati da una loro codifica affinché possano essere riconosciuti dai vari analizzatori (**#define EQ 258**). Di proposito si è parlato di analizzatori al plurale perché i token vengono passati in input all’analizzatore sintattico, che invocherà quello lessicale ogni volta che dovrà interpretare un simbolo; in aggiunta la memorizzazione del valore effettivo di un token è indispensabile per l’analisi semantica.

Il generatore automatico di analizzatori lessicali (lexer) più usato è *Lex*, un tool di Unix. Riceve in ingresso un file con estensione .lex, lo compila e ottiene un file.c; poi il file.c, che deve essere compilato a sua volta, produce in output l’analizzatore lessicale vero e proprio, detto anche lexer.

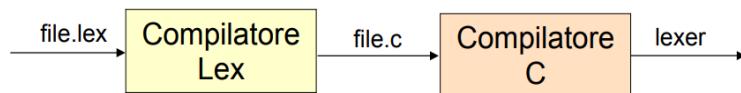


Figura 3.6: Schema di compilazione di un file.lex

La specifica di un programma Lex comprende tre sezioni:

Dichiarazioni

%%

Regole di traduzione

%%

Funzioni ausiliarie

Le *Dichiarazioni* sono di due tipologie: blackbox e whitebox. Quelle blackbox sono opzionali e comprendono l'inclusione di librerie o file esterni e la definizione di variabili e costanti. Quelle whitebox sono obbligatorie e rappresentano delle definizioni regolari, le quali non sono altro che espressioni regolari. Le *Regole di traduzione* specificano un'azione associata ai token e determinano quali stringhe lessicali sono riconosciute e quali invece no, causando la terminazione del programma con un errore. Infine nella sezione delle *Funzioni ausiliarie* si possono dichiarare un numero arbitrario di funzioni, anche nessuna, scritte in C, e che vengono richiamate dalle azioni all'interno delle regole di traduzione, eccezion fatta per il main se presente.

Lex è trattato più in dettaglio nell'Appendice A.

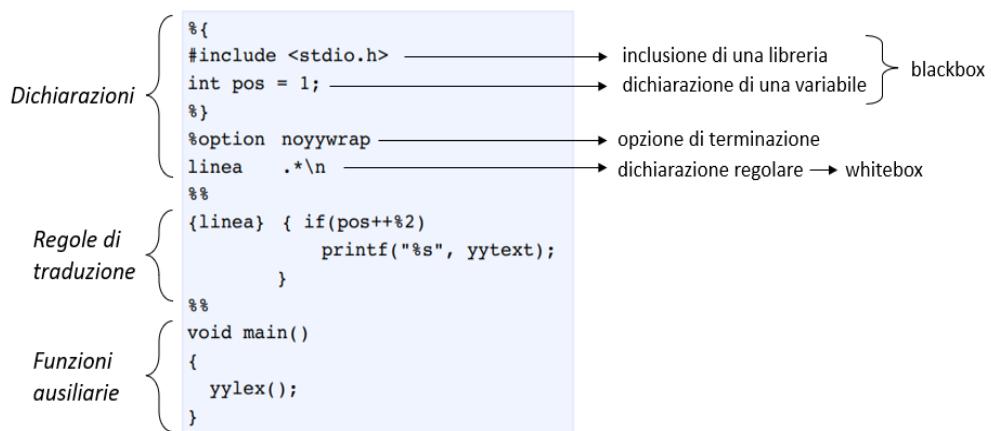


Figura 3.7: Programma Lex che stampa le linee di un file in posizione dispari

3.2.2 Analisi sintattica

L'analisi sintattica, detta anche parsing, assegna alle frasi del linguaggio una struttura che prende in considerazione i token ricevuti in input e le relazioni tra loro. Per svolgere tale compito, l'analizzatore prende come riferimento la grammatica BNF (o EBNF) che è stata definita per il linguaggio (sezione 2.2); inoltre, dato che l'analisi

sintattica e quella lessicale sono fortemente intrecciate, ogni espressione regolare può essere espressa tramite una grammatica. Lo strumento tramite cui opera è la derivazione che consiste nella costruzione top-down (dall'alto verso il basso) dell'albero sintattico e nella definizione delle regole di riscrittura. La derivazione può essere canonica sinistra o destra, in base alla posizione del nonterminale da sostituire ad ogni passo (più a sinistra o più a destra): in ogni caso l'albero sintattico che la rappresenta è sempre lo stesso. Esempio:

Produzione BNF di riferimento: $E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id$

Frase da derivare: $id + id * id$

Derivazione canonica a sinistra:

$$E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + id * E \Rightarrow id + id * id$$

Derivazione canonica a destra:

$$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * id \Rightarrow E + id * id \Rightarrow id + id * id$$

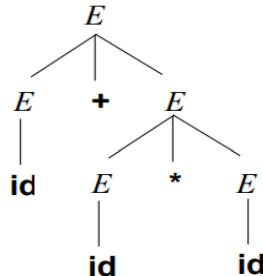


Figura 3.8: Albero sintattico della frase $id + id * id$

Una grammatica si dice ambigua quando esiste una frase associata a più alberi sintattici e quindi possiede molteplici derivazioni canoniche. Il caso illustrato sopra ne costituisce un esempio.

Derivazione canonica a sinistra alternativa:

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$$

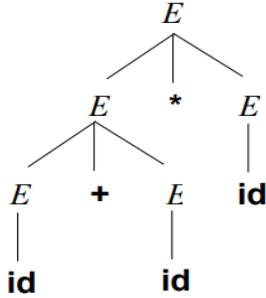


Figura 3.9: Albero sintattico alternativo della frase `id + id * id`

In linea generale si tende ad evitare l'uso di grammatiche ambigue per fare in modo che la sintassi del linguaggio risulti più semplice e chiara, tuttavia è possibile usufruirne: basta definire delle regole di disambiguamento per mezzo delle quali si riescano a scartare gli alberi alternativi. Un albero si definisce concreto quando si costruisce a partire dalla BNF, astratto quando si basa sulla EBNF.

La BNF (o EBNF) fa ampio utilizzo della tecnica della ricorsione che può essere diretta o indiretta. Si dice diretta quando un nonterminale richiama se stesso nella definizione della regola, mentre si dice indiretta quando un nonterminale richiama uno o più nonterminali, che vengono definiti successivamente. Una grammatica si definisce ricorsiva a sinistra se nella dichiarazione della regola il/i nontermianale/i si trovano a sinistra del/i terminale/i; si definisce ricorsiva a destra se accade l'opposto. Una grammatica ricorsiva a sinistra si può sempre trasformare in una grammatica ricorsiva a destra. Esempi:

$$A \rightarrow A\alpha \mid \beta \quad \Rightarrow \quad A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

Il primo esempio è una grammatica ricorsiva a sinistra e diretta, invece il secondo ne rappresenta la conversione in una ricorsiva a destra. Inoltre è indiretta su A , ma diretta rispetto ad A' . Si noti che durante il processo di trasformazione si devono utilizzare uno o diversi simboli nonterminali in più (A' non serve nel primo esempio).

Gli analizzatori sintattici, detti anche parser, possono essere di due tipologie: *top-down* o *bottom-up*. L’approccio top-down attua il processo di derivazione generando l’albero a partire dalla sua radice (assioma) e fino ad arrivare alle foglie (i terminali): logicamente e graficamente parlando procede dall’alto verso il basso. La metodologia più impiegata è il parsing LL(1): la prima L significa *Left to right scanning*, ovvero lettura da sinistra a destra; la seconda L sta per *Leftmost derivation*, cioè derivazione canonica a sinistra; il numero 1 indica che necessita di *un simbolo di lookahead*, che è il prossimo token in input, per fare un’analisi predittiva e decidere quale azione intraprendere. Lo sfruttamento di un singolo simbolo di lookahead ha permesso l’automatizzazione dell’analisi sintattica. Il programma di parsing prende in ingresso tutti i simboli grammaticali, memorizzati su una struttura a pila, e li confronta con l’input, che è la sequenza dei terminali. Sia **X** un token ed **a** un terminale: se **X** = **a** = \$ (carattere speciale usato dal parser), allora la frase viene accettata; se **X** è un nonterminale e **X** = **a** ≠ \$, allora si avanza e si sostituisce **X** secondo le regole della tabella di parsing; se **X** è un terminale e **X** = **a**, allora si ha un matching e il token in questione viene rimosso sia dal prossimo elemento della pila sia dall’input; se **X** ≠ **a** si ha un errore e si esce dal programma. Se il linguaggio richiede che ogni comando termini con il carattere “;”, un esempio di errore sintattico è sua omissione alla fine della riga. Il parser top-down è inefficiente quando si ha a che fare con una grammatica ricorsiva a sinistra.

$$S \rightarrow (S)S \mid \epsilon$$

| | | | |
|---|----------------------|--------------------------|--------------------------|
| | (|) | \$ |
| S | $S \rightarrow (S)S$ | $S \rightarrow \epsilon$ | $S \rightarrow \epsilon$ |

(Tabella di parsing)

$$w = () \quad (\text{Frase, cioè una stringa di terminali})$$

| Pila | Input | Azione | |
|-------|-------|--------------------------|----------|
| S\$ | (\$ | $S \rightarrow (S)S$ | (avanza) |
| (S)\$ |) \$ | match | |
| S)S\$ |) \$ | $S \rightarrow \epsilon$ | |
|)S\$ |) \$ | match | |
| S\$ | \$ | $S \rightarrow \epsilon$ | |
| \$ | \$ | accetta | |

Figura 3.10: Esempio di parsing top-down

Il parsing bottom-up agisce esattamente in maniera opposta rispetto a quello top-down: genera l'albero a partire dalle sue foglie e infine deriva la radice; logicamente e graficamente parlando procede dal basso verso l'alto. Risulta maggiormente versatile siccome non ha nessun problema con la grammatica ricorsiva a sinistra. La metodologia più usata è il parsing LALR(1): LA indica che guarda il simbolo di *LookAhead*; la L significa *Left to right scanning*; R sta per *Rightmost derivation*, cioè derivazione canonica a destra. Il suo funzionamento è analogo a quello descritto per il parsing top-down, dove l'azione *sposta* è l'avanzamento dell'input. L'unica differenza è l'azione di *riduzione*, che attua una derivazione inversa, ovvero dalla frase concreta risale all'assioma.

$$\begin{array}{l} S' \rightarrow S \\ S \rightarrow (S)S \mid \epsilon \end{array}$$

$$\text{stringa} = ()$$

| Pila | Input | Azione | |
|---------|-------|--------------------------|-------------|
| \$ | (\$ | sposta | (riduzione) |
| \$(|) \$ | $S \rightarrow \epsilon$ | |
| \$(\$ |) \$ | sposta | |
| \$(\$) | \$ | $S \rightarrow \epsilon$ | |
| \$(\$)S | \$ | $S \rightarrow (S)S$ | |
| \$S | \$ | $S' \rightarrow S$ | |
| \$S' | \$ | accetta | |

Figura 3.11: Esempio di parsing bottom-up

Il generatore automatico di parser più diffuso è *Yacc* (Yet Another Compiler Compiler), che effettua un'analisi di tipo LALR(1) sfruttando una grammatica ricorsiva a sinistra non ambigua ed è un tool della famiglia Unix. Viene usato sempre insieme a Lex: infatti i file Yacc, riconoscibili dall'estensione .y, sono compilati allo stesso modo di quelli Lex e la specifica di Yacc è strutturalmente identica a quella di Lex. Cambiano solo i significati delle dichiarazioni whitebox e delle regole di traduzione. Le dichiarazioni whitebox rappresentano i token ricevuti in ingresso dal lexer; le regole di traduzione definiscono la grammatica vera e propria del linguaggio, grazie alla quale si può ricavare l'albero sintattico. Per approfondire l'argomento si consulti l'Appendice B.

```
line → expr eol
expr → expr + term | term
term → term * factor | factor
factor → ( expr ) | digit
```

(BNF del linguaggio)

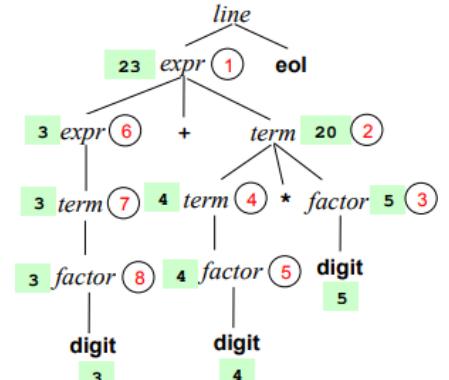
(Regole di traduzione)

```
%{
#include <stdio.h>
#include <ctype.h>
int yylex();
void yyerror();
%}
%token DIGIT
%%
line : expr '\n' { printf("%d\n", $1); }
expr : expr '+' term { $$ = $1 + $3; }
      | term { $$ = $1; }
term : term '*' factor { $$ = $1 * $3; }
      | factor { $$ = $1; }
factor : '(' expr ')' { $$ = $2; }
       | DIGIT { $$ = $1; }
%%
int yylex()
{ int c;
  c = getchar();
  if (isdigit(c)){
    yylval = c - '0';
    return(DIGIT);
  }
  return(c);
}

void yyerror(){fprintf(stderr, "Syntax error\n");}
void main(){yparse();}
```

Frase del linguaggio

3 + 4 * 5



Albero sintattico concreto generato da Yacc e che rappresenta la frase

Figura 3.12: Esempio di programma Yacc

3.2.3 Analisi semantica

L’analisi semantica verifica che le frasi sintatticamente corrette di un linguaggio siano di senso compiuto e computa informazione aggiuntiva rispetto al parser prima dell’esecuzione: per questo viene considerata statica. A causa della mancanza di uno standard della semantica di un linguaggio (vedi sezione 2.2), non esiste né un generatore di analizzatori semantic automatico, come Lex o Yacc, né degli approcci consolidati, come il parsing top-down o bottom-up. Se la specifica del lessico e della sintassi sono fornite dal progettista del linguaggio, quella della semantica è a carico del progettista del compilatore (o dell’interprete o del traduttore). Inoltre la qualità dell’analisi dipende dalla staticità di un linguaggio: più è alta e più è sicuro, cioè minimizza la probabilità che ci siano errori durante l’esecuzione. Per staticità si intende quanto fortemente sia tipizzato un linguaggio: Lisp → C → Pascal → Ada hanno staticità crescente.

L’analizzatore semantico riceve in input l’albero sintattico e deve decidere che ordine di scorrimento seguire: se partire dalle foglie e risalire fino alla radice o se procedere dall’alto verso il basso. A questo punto attraversa l’albero nodo per nodo, svolgendo l’azione associata a ciascuno di essi, che può essere di natura più o meno complessa. Per scorrere una struttura gerarchica in modo ottimale, conviene utilizzare i puntatori di C o un costrutto simile, facendo puntare il nodo analizzato a quello successivo, che può essere un “fratello”, cioè un nodo di pari livello di quello attuale (appartenente alla stessa produzione BNF) oppure un “figlio”, ovvero un nodo generato da quello attuale (appartenente alla produzione BNF seguente).

Prendendo come riferimento l’albero in figura 3.12, scegliendo un attraversamento dalla radice alle foglie e ipotizzando che il linguaggio, oltre alle cifre, possa comprendere anche le lettere (astratte dal simbolo **letter**), possiamo fare alcuni esempi dei

passaggi dell’analizzatore semantico:

- Per il sotto albero $expr \rightarrow term \rightarrow factor \rightarrow \text{digit}$ l’azione di ogni nodo consiste semplicemente nel collegarsi a quello successivo, tranne per $factor$, che ha anche il compito di controllare a quale terminale è associato: in questo caso a **digit**, ma poteva anche essere il token **letter**.
- Per il sotto albero $term * factor$, una volta determinato a quali terminali corrispondono i singoli nodi, bisogna verificare che siano compatibili con l’operazione $*$: in questo caso sia $term$ che $factor$ sono associati al token **digit** e quindi la semantica risulta corretta dato che due cifre possono essere moltiplicate fra loro (**4 * 5**). Se però $factor$ corrispondesse al simbolo **letter**, allora il programma segnalerebbe un errore (è impossibile svolgere il prodotto tra una lettera e una cifra).

Il caso appena illustrato evidenzia che il *type checking*, ovvero il controllo della correttezza dei tipi delle espressioni/istruzioni del linguaggio, è uno dei ruoli principali dell’analisi semantica. Tale procedimento è indispensabile, in quanto non ogni tipo di dato è adatto a ciascuna istruzione permessa dal linguaggio: per chiarire meglio il concetto facciamo degli esempi.

- Le operazioni aritmetiche binarie ($+$, $-$, $*$, \div) richiedono due tipi numerici, che siano interi o reali, e non sono compatibili con nessun altro tipo (booleano, carattere);
- L’espressione condizionale *if-then-else* ammette solo un tipo/espressione booleana dopo l’*if* ed inoltre le espressioni/variabili/costanti dichiarate dopo il *then* e l’*else* devono essere dello stesso tipo;

- L’assegnamento nella forma $x = expr$, richiede che il tipo di x e di $expr$, che può essere un’espressione/variabile/costante, siano identici, altrimenti è ritenuto un’operazione illecita.

L’informazione sul tipo di un dato viene recuperata dall’albero sintattico, ma è memorizzata anche all’interno di una struttura denominata *Symbol Table*: un catalogo degli identificatori, che sono associati al correlativo tipo. A seconda della classificazione di un linguaggio (sezione 2.1), gli identificatori sono di vario genere: variabili, funzioni, strutture, classi, metodi, eccetera. La *Symbol Table* viene creata per non permettere la dichiarazione di due identificatori duplicati: due variabili denominate y o una funzione con lo stesso nome di una già esistente. Altre informazioni utili da memorizzare potrebbero essere lo spazio allocato o la visibilità (globale o locale) degli identificatori. Vedremo in seguito che modo il linguaggio Tofu utilizza la *Symbol Table*.

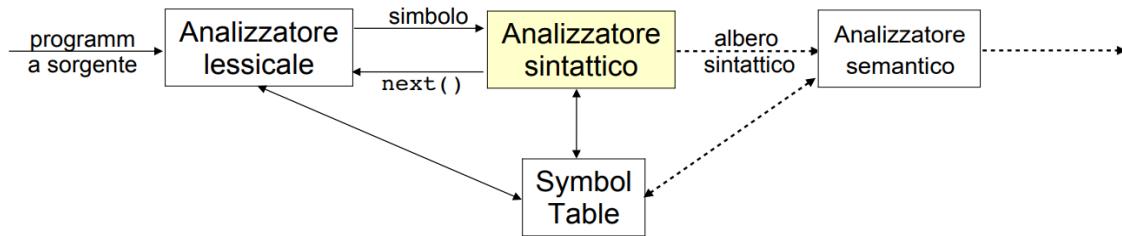


Figura 3.13: Schema riassuntivo dell’analisi di un linguaggio

L’output dell’analizzatore semantico viene passato al generatore di codice intermedio, insieme all’albero sintattico, sia che si tratti di un generatore interno al compilatore che di un generatore appartenente al traduttore e che quindi produce codice intermedio visibile all’esterno. Nel caso di un interprete, l’output dell’analisi semantica e l’albero sintattico sono usati per interpretare direttamente il sorgente.

4 Il linguaggio di programmazione Tofu

Tofu, abbreviazione di **Toy functions**, è un linguaggio ibrido fra il paradigma imperativo e quello funzionale; inoltre è di tipo pseudocompilato, dato che per la sua implementazione si è scelto di scrivere un traduttore più una macchina virtuale. Un programma Tofu è strutturato in tre sezioni: variabili, funzioni e corpo; le prime due sono opzionali, mentre la terza è obbligatoria. Nella prima sezione, introdotta dalla keyword **variables**, vengono dichiarate le variabili globali del programma, che possono essere di tipo atomico *intero*, *stringa* o *booleano*, oppure del tipo composto *lista*. Una lista si dice semplice o di primo livello se al suo interno sono presenti solo dati atomici, mentre si dice innestata o a più livelli se contiene altre liste, che a loro volta possono essere innestate o semplici, fino ad arrivare alla lista più interna, che deve essere di primo livello. Una lista può essere anche vuota. Nella seconda sezione, introdotta dalla keyword **functions**, sono dichiarate le funzioni del programma che devono restituire un dato atomico o una lista: dunque non possono essere di tipo *void*. Inoltre il corpo della funzione è una singola espressione senza effetti collaterali: questo significa che vengono referenziate solo variabili locali. Le funzioni possono essere chiamate all'interno dell'ultima sezione, da altre funzioni o anche da loro stesse in modo ricorsivo. Infine il corpo del programma, introdotto dalla keyword **body**, è una sequenza di istruzioni che sono di due tipologie: l'assegnamento di una variabile o la stampa a video del valore di un'espressione. Il paradigma imperativo risulta più evidente nella sezione delle variabili e in quella del corpo del programma, mentre quello funzionale nella sezione delle funzioni e nell'uso del tipo lista. Inoltre entrambi i paradigmi influenzano il controllo di un programma Tofu, che è governato da istruzioni condizionali e dalla ricorsione. Un commento in Tofu è introdotto dal

metacarattere #.

Vediamo ora i dettagli del linguaggio di programmazione Tofu.

Dichiarazione dei tipi atomici e il loro assegnamento:

- **int** naturale: int;

...

naturale = 20;

- **string** parola: string;

...

parola = "casa";

- **bool** ok: bool;

...

ok = true;

Dichiarazione del tipo lista e suo assegnamento:

- lista semplice primi: [int];

...

primi = [2, 3, 5, 7, 11, 13, 17];

- lista innestata nested: [[[string]]];

...

nested = [[[“a”, “b”], [“c”]], [[“d”, “e”, “f”]]];

- lista vuota controlli: [bool];

...

controlli = [];

- lista costruita strutturata: [int];
- tramite ...
- espressioni strutturata = [3*(10-1), i/j, max(i, k), min(j, x), x-y+8];

Tofu prevede diversi tipi di espressioni. Quelle aritmetiche sono costituite dalle quattro operazioni elementari, indicate dagli operatori `+`, `-`, `*`, `/`, e applicabili solo al tipo **int**. L'operatore `-` è usato anche per il cambiamento di segno. Le espressioni di uguaglianza, `==`, e disuguaglianza, `!=`, sono applicabili a tutti i tipi, mentre le altre espressioni di confronto, `>`, `>=`, `<`, `<=`, sono applicabili solo ai tipi **int** e **string**. Il risultato di qualsiasi operazione di confronto è un tipo **bool**. Infine gli operatori logici **and**, **or** e **not**, indicati coi loro stessi nomi, vengono valutati in corto circuito e sono applicabili al solo tipo **bool**.

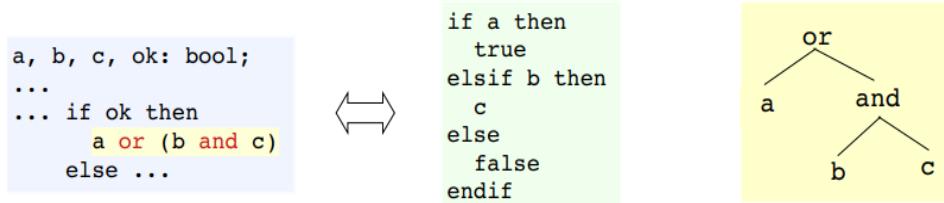


Figura 4.1: Esempio di espressioni logiche valutate in corto circuito

L'ordine di valutazione degli operandi va da sinistra a destra, mentre per quanto riguarda l'associatività e la precedenza degli operatori:

| Operatore | Tipo | Associatività |
|---|---------|---------------|
| and, or | binario | sinistra |
| <code>==</code> , <code>!=</code> , <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> | binario | nonassoc |
| <code>+</code> , <code>-</code> , <code>++</code> | binario | sinistra |
| <code>*</code> , <code>/</code> | binario | sinistra |
| <code>-</code> , <code>not</code> | unario | destra |

precedenza crescente

Figura 4.2: Tabella che indica l'associatività e la precedenza degli operatori

In Tofu esiste anche l'operazione di *range* applicabile al solo tipo **int** e che restituisce la *lista* dei numeri consecutivi compresi tra due estremi:

```

range: [int];
...
range = [21..30];
output: range = [21,22,23,24,25,26,27,28,29,30]

```

Una funzione viene dichiarata nel seguente modo:

```

isEqual(str: string) → bool
str == "alpha";

```

dove *isEqual* è il nome della funzione, *str* è un suo parametro (variabile locale), **bool** è il tipo restituito e il resto è il corpo della funzione.

In Tofu esistono sei funzioni built-in (già presenti all'interno del linguaggio senza che il programmatore debba definirle) applicabili al solo tipo *lista*:

- La *empty* controlla se una lista è vuota, restituendo **true** in caso affermativo, **false** altrimenti.


```

isVuota1, isVuota2: bool;
...
isVuota1 = empty([]);
isVuota2 = empty([true,false]);
output: isVuota1 = true, isVuota2 = false

```
- La *length* restituisce la lunghezza di una lista, cioè il numero dei suoi elementi.


```

dim: [int];
...
dim = length([-1, -3, -5, -7, -9]);
output: dim = 5

```
- L'operazione di append, indicata da *++*, prende in ingresso due liste e ne restituisce la concatenazione.


```

app: [string];
...
app = ["Un", "esempio"] ++ ["di", "append"];
output: app = ["Un", "esempio", "di", "append"]

```

- La *head* restituisce il primo elemento di una lista, il quale potrebbe essere un tipo atomico o a sua volta una lista.
- La *tail*, complementare della *head*, restituisce la lista privata del suo primo elemento.
- La *last* restituisce l'ultimo elemento di una lista, il quale potrebbe essere un tipo atomico o a sua volta una lista.
- La *init*, complementare della *last*, restituisce la lista privata del suo ultimo elemento.

```

testa: int;
...
testa = head([41, 31, 21, 11, 1]);
output: testa = 41

coda: [int];
...
coda = tail([41, 31, 21, 11, 1]);
output: coda = [31, 41, 21, 11, 1]

ultimo: [string];
...
ultimo = last([[“giallo”],[],[“rosso”, “blu”]]);
output: ultimo = [“rosso”, “blu”]

inizio: [[string]];
...
inizio = init([[“giallo”],[],[“rosso”, “blu”]]);
output: inizio = [[“giallo”],[]]

```

Le funzioni *head*, *last*, *tail*, *init* sono polimorfe: a seconda della lista in input, producono in output un tipo di dato differente. Ad esempio la *head*([60, 70, 80]) restituisce l'intero 60, mentre la *head*([[60, 70, 80], [], [1000]]) restituisce la lista semplice [60, 70, 80].

Il linguaggio ha due tipologie di espressioni condizionali: l'*if-else* e le *guardie*. L'*if-else* si esprime nel seguente modo:

if *expr*₁ **then** *expr*₂ **else** *expr*₃ **end**

dove *expr*₁ deve essere di tipo **bool**, mentre *expr*₂ ed *expr*₃ possono essere di qual-

siasi tipo: l'importante è che sia lo stesso per entrambe le espressioni. Esempio:

```
a,b,c: int;  
...  
a = if b <= c then b+7 else (c-3)*9 end;
```

Le *guardie* sono utili per esprimere molteplici alternative in una forma compatta, anziché dover scrivere una cascata di *if-else* nidificati:

```
| cond1 → expr1;  
| cond2 → expr2;  
...  
| otherwise → exprn
```

dove le *cond_i* sono espressioni booleane, mentre le *expr_i* sono le espressioni che rappresentano il valore computato nel caso la condizione sia vera e che possono essere di qualunque tipo, a patto che sia identico per tutte. Quindi se *cond₁* risulta vera, viene restituita *expr₁*, altrimenti si verifica *cond₂*, che se è vera, restituisce *expr₂*; in caso contrario si continua in questa maniera fino a restituire *expr_n* se tutte le *cond_i* sono false. Esempio:

```
...  
functions  
asciiABC(codice:int)→ string  
| codice == 65 → “A”;  
| codice == 66 → “B”;  
| codice == 67 → “C”;  
| otherwise → “Codice non valido”;  
...
```

Come tutti i linguaggi, anche Tofu permette di interagire con l'utente facendogli inserire degli input da tastiera durante l'esecuzione di un programma. Tale compito

viene svolto dall'istruzione

?(type)

dove *type* è un'astrazione che indica tutti i tipi. Esempi:

| | | |
|---------------------|--------------------------------|------------------------|
| nomi: [string] | massimo: int; | ok, b1, b2: bool; |
| ... | ... | ... |
| nomi = ?([string]); | massimo = max(?(int), ?(int)); | b1 = ?(bool); |
| | | b2 = ?(bool); |
| | | ok = b1 and b2; |

Per il display di un valore di una variabile ci sono due costrutti:

!(expr) **show expr**

dove *expr* indica una qualsiasi espressione ammissibile in Tofu. Il **!** mostra a video il valore dell'espressione in maniera dinamica. L'istruzione **show** viene usata solo nel **body** del programma, mentre il **!** può anche essere usato all'interno della sezione **functions**. Esempi:

| | |
|-----------------------------------|----------------------------|
| ... | n: int; |
| functions | ... |
| max(a: int, b: int) → int | n = ?(int); |
| if !(a) > !(b) then a else b end; | show "Il fattoriale di " ; |
| ... | show n; |
| body | show " è "; |
| show "Hello world!"; | show fattoriale(n); |

Di seguito vengono mostrati due esempi di programmi Tofu completi.

```

1  variables
2      min, max: int;
3      lista: [int];
4
5  functions
6      minimo(lista: [int]) -> int
7          if empty(tail(lista)) then head(lista)
8          else if head(lista) <= minimo(tail(lista)) then head(lista)
9          else minimo(tail(lista))
10         end
11     end;
12
13     massimo(lista: [int]) -> int
14         if empty(tail(lista)) then head(lista)
15         else if head(lista) >= massimo(tail(lista)) then head(lista)
16         else massimo(tail(lista))
17         end
18     end;
19
20 body
21     show "Questo programma indica l'elemento minore e quello maggiore di una lista di interi.\n";
22     show "Inserisci una lista di interi: ";
23     lista = ?([int]);
24     min = if empty(lista) then 0 else minimo(lista) end;
25     show "Il minimo è: ";
26     show min;
27     show "\n";
28     max = if empty(lista) then 0 else massimo(lista) end;
29     show "Il massimo è: ";
30     show max;

```

Figura 4.3: Programma Tofu che identifica l'intero minore e quello maggiore all'interno di una lista (max_min.tofu)

```

1  variables
2      listaInput, invertita: [[string]];
3      msgInserimento: string;
4
5  functions
6      invertiSingolaLista(lista:[string]) -> [string]
7          | empty(lista) -> [];
8          | otherwise -> [last(lista)] ++ invertiSingolaLista(init(lista));
9
10     inverti(daInvertire: [[string]]) -> [[string]]
11        | empty(daInvertire) -> [];
12        | otherwise -> [invertiSingolaLista(last(daInvertire))] ++ inverti(init(daInvertire));
13
14 body
15     show "Questo programma inverte una lista di liste di stringhe.\n";
16     msgInserimento = "Inserisci una lista di stringhe con profondità 2: ";
17     show msgInserimento;
18     listaInput = ?([[string]]);
19     invertita = inverti(listaInput);
20     show "Lista invertita = ";
21     show invertita;

```

Figura 4.4: Programma Tofu che inverte una lista di liste di stringhe (inverti_liste.tofu)

5 Il traduttore di Tofu

Come detto nel capitolo precedente, Tofu è un linguaggio pseudocompilato: dunque prevede un traduttore e una macchina virtuale (o interprete). In questo capitolo si analizzerà l'implementazione del traduttore a partire dall'analizzatore lessicale e fino ad arrivare al generatore di codice intermedio. Nel prossimo capitolo ci si concentrerà sulla macchina virtuale.

Innanzitutto si è definito un file con estensione .h, chiamato *def.h*, in cui sono state dichiarate le costanti, le variabili e le funzioni di base, che vengono usate sia nel traduttore che nell'interprete.

```

1  #include <stdio.h>          35   NIF_EXPR,           68   LIST,            102  PUTS,
2  #include <stdlib.h>          36   NINPUT,            69   LOAD,            103  T_SHOW
3   typedef enum                37   NOUTPUT,           70   STOR,            104  } Operator;
4  {                            38   NFUNC_CALL,        71   SKIP,            105
5   NPROGRAM,                  39   NBODY_SECTION,    72   EQUA,            106  typedef union
6   NVAR_SECTION,              40   NSTAT_LIST,       73   NEQU,            107  {
7   NVAR_DECL,                 41   NSTAT,             74   GRTR,            108  int ival;
8   NID_LIST,                  42   NASSIGN_STAT,    75   GREQ,            109  char *sval;
9   NTYPE,                     43   NSHOW_STAT,      76   LETH,             110  } Value;
10  NLIST_TYPE,                44 } Nonterminal;     77   LEQ,             111
11  NFUNC_SECTION,             45   typedef enum       78   PLUS,            112  typedef struct snode
12  NFUNC_DECL,                46 {                   79   MINU,            113  {
13  NOPT_FORMAL_LIST,          47   {                   80   MULT,            114  Typenode type;
14  NFORMAL_DECL,              48   T_INT,             81   DIVI,            115  Value value;
15  NGUARD_EXPR,               49   T_STRING,          82   APPN,            116  struct snode *p1 ,*p2, *p3;
16  NGUARD_LAST,               50   T_BOOL,            83   UMIN,            117  } Node;
17  NAND,                      51   T_INCONST,         84   NEGA,             118
18  NOR,                       52   T_BOOLCONST,      85   RANG,             119  typedef Node *Pnode;
19  NEQ,                        53   T_STRCONST,       86   T_EMPT,          120
20  NNE,                        54   T_ID,              87   T LENG,           121  char *newstring(char*);
21  NGR,                        55   T_NONTERMINAL,    88   T_HEAD,          122
22  NGE,                        56 } Typenode;        89   T_TAIL,          123  int yylex();
23  NLS,                         57   typedef enum {      90   T_LAST,          124
24  NLE,                        58   VARS,             91   T_INIT,          125  Pnode nontermnode(Nonterminal),
25  NPLUS,                      59   HALT,             92   PUSH,            126  idnode(),
26  NMINUS,                     60   NEWI,             93   JUMP,            127  keynode(Typenode),
27  NMULT,                      61   NEWS,             94   APOP,            128  intconstnode(),
28  NDIV,                       62   NEWB,             95   T_FUNC,          129  strconstnode(),
29  NAPP,                       63   NEWL,             96   RETN,            130  boolconstnode(),
30  NNEG_MINUS,                64   LOCI,             97   GETI,            131  newnode(Typenode);
31  NNOT,                       65   LOCB,             98   GETS,            132
32  NRANGE,                     66   LOCS,             99   GETB,            133  void treeprint(Pnode, int),
33  NOPT_EXPR_LIST,             67   LOCS,            100  GETL,            134  yyerror(), parser();

```

Figura 5.1: Il codice C del file *def.h*

L'enum *Nonterminal* identifica i simboli nonterminali dell'albero sintattico all'interno del traduttore; invece l'enum *Typenode* indica di che tipologia è un nodo dell'albero, riconoscendo il terminale esatto (int, costante booleana, id, ecc...) o dicendo semplicemente che è un generico nonterminale. L'enum *Operator* specifica tutte le istruzioni del codice intermedio di Tofu (argomento trattato nella sezione 5.4). La union *Value* serve a memorizzare il valore lessicale di un terminale, che può essere intero o stringa; viene usato all'interno della struct *Node*, che rappresenta un nodo dell'albero. Un nodo dunque è caratterizzato dal suo tipo (*Typenode*), dal suo valore lessicale (*Value*) e dal suo collegamento con altri nodi: **p1* è un puntatore al primo figlio, **p2* al secondo e **p3* punta al fratello destro. Nel traduttore, per costruire l'albero sintattico, si utilizza il puntatore al nodo **Pnode*. Infine nel *def.h* vengono dichiarate le funzioni che verranno implementate dall'analizzatore lessicale e sintattico.

5.1 Il lexer

Viene illustrato nella figura sottostante l'analizzatore lessicale di Tofu.

```

1  %}
2  #include "parser.h"
3  #include "def.h"
4  int line = 1;
5  Value lexval;
6  %}
7
8  %option noyywrap
9
10 spacing   ([ \t])+
11 letter    [A-Za-z]
12 digit     [0-9]
13 intconst  {digit}+
14 strconst  \"([^\"])*\""
15 boolconst false|true
16 id        {letter}{letter}|{digit})*
17 op        [+|-/*<>=]
18 comment   #.*"
19 sugar     [:;,()\\[]!\\?.\\|]
20
21 %%
22
23 {spacing} ;
24 {comment} ;
25 \n        {line++;}
26 {sugar}   {return(yytext[0]);}
27 variables {return(VARIABLES);}
28 functions {return(FUNCTIONS);}
29 if         {return(IF);}
30 then      {return(THEN);}
31 else      {return(ELSE);}
32 end       {return(END);}
33 otherwise {return(OTHERWISE);}
34 body      {return(BODY);}

35 show      {return(SHOW);}
36 "->"    {return(RETURN);}
37 and      {return(AND);}
38 or       {return(OR);}
39 "=="    {return(EQ);}
40 "!="    {return(NE);}
41 ">="    {return(GE);}
42 "<="    {return(LE);}
43 {op}      {return(yytext[0]);}
44 "+"     {return(APP);}
45 "not"   {return(NOT);}
46 int     {return(INT);}
47 string  {return(STRING);}
48 bool    {return(BOOL);}
49 {intconst} {lexval.ival = atoi(yytext);
50           | return(INTCONST);}
51 {strconst} {lexval.sval = newstring(yytext);
52           | return(STRCONST);}
53 {boolconst} {lexval.ival = (yytext[0] == 'f' ? 0 : 1);
54           | return(BOOLCONST);}
55 {id}      {lexval.sval = newstring(yytext);
56           | return(ID);}
57 .        {return(ERROR);}
58
59 %%
60
61 char *newstring(char *s)
62 {
63     char *p;
64
65     p = malloc(strlen(s)+1);
66     strcpy(p, s);
67     return(p);
68 }
```

Figura 5.2: Il codice del file *lexer.lex*

I caratteri di tab e di ritorno a capo sono indicati come in C (\t e \n); la variabile *line* viene aggiornata ad ogni nuova riga del programma Tofu (riga 10 della figura 5.2), in modo da poter individuare la linea esatta se si verifica un errore. Una costante stringa si deve dichiarare fra i doppi apici “”; un identificatore inizia con una lettera e può opzionalmente essere seguito da altre lettere o da cifre. Si noti che il carattere \ nelle dichiarazioni whitebox ha il compito di inibire i metacaratteri: ad esempio \. (riga 19 della figura 5.2) indica il semplice carattere punto (usato per l'operazione di range) e non il metacarattere che identifica qualsiasi simbolo non presente nelle regole

di traduzione (riga 57). All'interno di queste ultime, alle keyword e alle stringhe lessicali di Tofu viene associato un valore di ritorno univoco: ad esempio **if** con **IF** (riga 29) o “**++**” con **APP** (riga 44). Ci sono eccezioni come gli spazi e i commenti che, dichiarando una regola vuota, vengono ignorati (righe 23 e 24) o gli zuccheri lessicali e i singoli operatori, che semplicemente restituiscono loro stessi (righe 26 e 43). Inoltre nelle regole di traduzione delle costanti e dell'identificatore, prima di restituire il token, il loro valore lessicale viene salvato:

- Il valore dell'*intconst* viene trasformato da un array di caratteri (stringa) a un intero, grazie alla funzione C *atoi()* e memorizzato come tale.
- Il valore della *boolconst* se inizia col carattere ‘**f**’, significa che nel programma Tofu è **false**, e dunque gli viene associata la cifra **0** (corrispondente a falso in C); altrimenti è **true** e vale **1** in C.
- La *strconst* e l'*id* memorizzano il loro valore lessicale come una stringa, grazie alla funzione *newstring()*, dichiarata nella sezione delle funzioni ausiliarie di Lex, che si occupa di allocare la memoria necessaria per salvare la stringa.

Se in un programma viene inserito il carattere **%**, non riconosciuto da Tofu, si ha un'errore lessicale e viene stampato il seguente messaggio:

```
dario01@LAPTOP-RKFL0J0J:~/Tofu$ ./T_code <programs/inverti_liste.tofu
Line 7: error on symbol "%"
```

Figura 5.3: Esempio di errore lessicale in Tofu

5.2 Il parser

Viene mostrato di seguito l'analizzatore sintattico di Tofu.

```

71 expr_decl : guard_expr_list
72 | | | | | expr
73
74 guard_expr_list : guard_expr guard_expr_list {$$ = $1; $1->p3 = $2;}
75 | | | | guard_expr guard_last {$$ = $1; $1->p3 = $2;}
76 | |
77
78 guard_expr : '|' expr RETURN expr ';' {$$ = nontermnode(NGUARD_EXPR); $$->p1 = $2; $$->p2 = $4;}
79
80 guard_last : '|' OTHERWISE RETURN expr {$$ = nontermnode(NGUARD_LAST); $$->p1 = $4;}
81
82
83 expr : expr bool_op bool_term {$$ = $2; $$->p1 = $1; $$->p2 = $3;}
84 | bool_term
85 | ;
86
87 bool_op : AND {$$ = nontermnode(NAND);}
88 | OR {$$ = nontermnode(NOR);}
89 | ;
90
91 bool_term : comp_term comp_op comp_term {$$ = $2; $$->p1 = $1; $$->p2 = $3;}
92 | comp_term
93 | ;
94
95 comp_op : EQ {$$ = nontermnode(NEQ);}
96 | NE {$$ = nontermnode(NNE);}
97 | '>' {$$ = nontermnode(NGR);}
98 | GE {$$ = nontermnode(NGE);}
99 | '<' {$$ = nontermnode(NLS);}
100 | LE {$$ = nontermnode(NLE);}
101 | ;
102
103 comp_term : comp_term add_op term {$$ = $2; $$->p1 = $1; $$->p2 = $3;}
104 | term
105 | ;
106
107 add_op : '+' {$$ = nontermnode(NPLUS);}
108 | '-' {$$ = nontermnode(NMINUS);}
109 | APP {$$ = nontermnode(NAPP);}
110 | ;
111
112 term : term mul_op factor {$$ = $2; $$->p1 = $1; $$->p2 = $3;}
113 | factor
114 | ;
115
116 mul_op : '*' {$$ = nontermnode(NMULT);}
117 | '/' {$$ = nontermnode(NDIV);}
118 | ;
119
120 factor : unary_op factor {$$ = $1; $$->p1 = $2;}
121 | '(' expr_decl ')' {$$ = $2;}
122 | ID {$$ = idnode();} {$$ = $2;}
123 | INTCONST {$$ = intconstnode();}
124 | STRCONST {$$ = strconstnode();}
125 | BOOLCONST {$$ = boolconstnode();}
126 | '[' opt_expr_list ']' {$$ = $2;}
127 | if_expr
128 | func_call
129 | input
130 | output
131 | range
132 | ;
133
134 unary_op : '-' {$$ = nontermnode(NNEG_MINUS);}
135 | NOT {$$ = nontermnode(NNOT);}
136 | ;
137
138 opt_expr_list : expr_list {$$ = nontermnode(NOPT_EXPR_LIST); $$->p1 = $1;}
139 | {$$ = nontermnode(NOPT_EXPR_LIST);}
140 | ;
141 expr_list : expr ',' expr_list {$$ = $1; $1->p3 = $3;}
142 | expr
143 | ;
144
145 if_expr : IF expr THEN expr ELSE expr END {$$ = nontermnode(NIF_EXPR); $$->p1 = $2; $2->p3 = $4; $4->p3 = $6;}
146 | ;

```

```

147 func_call : ID {$$ = idnode();} '(' opt_expr_list ')' {$$ = nontermnode(NFUNC_CALL); $$->p1 = $2; $$->p2 = $4;}
148 | | |
149 | | ;
150
151 input : '?' '(' type ')' {$$ = nontermnode(NINPUT); $$->p1 = $3;}
152 | | ;
153
154 output : '!' '(' expr_decl ')' {$$ = nontermnode(NOUTPUT); $$->p1 = $3;}
155 | | ;
156
157 range : '[' expr '.' '.' expr ']' {$$ = nontermnode(NRANGE); $$->p1 = $2; $$->p2 = $5;}
158 | | ;
159
160 body_section : BODY stat_list {$$ = nontermnode(NSTAT_LIST); $$->p1 = $2;}
161 | | |
162 | | ;
163
164 stat_list : stat stat_list {$$ = $1; $1->p3 = $2;}
165 | | stat
166 | | ;
167
168 stat : assign_stat
169 | | show_stat
170 | | ;
171
172 assign_stat : ID {$$ = idnode();} '=' expr_decl ';' {$$ = nontermnode(NASSIGN_STAT); $$->p1 = $2; $2->p3 = $4;}
173
174 show_stat : SHOW expr_decl ';' {$$ = nontermnode(NSHOW_STAT); $$->p1 = $2;}
175
176 %%
177
178 Pnode nontermnode(Nonterminal nonterm) {
179     Pnode p = newnode(T_NONTERMINAL);
180     p->value.ival = nonterm;
181     return(p);
182 }
183
184 Pnode idnode() {
185     Pnode p = newnode(T_ID);
186     p->value.sval = lexval.sval;
187     return(p);
188 }
189
190 Pnode keynode(Typenode keyword) {
191     return(newnode(keyword));
192 }
193
194 Pnode intconstnode() {
195     Pnode p = newnode(T_INCONST);
196     p->value.ival = lexval.ival;
197     return(p);
198 }
199
200 Pnode strconstnode() {
201     Pnode p = newnode(T_STRCONST);
202     p->value.sval = lexval.sval;
203     return(p);
204 }
205
206 Pnode boolconstnode() {
207     Pnode p = newnode(T_BOOLCONST);
208     p->value.ival = lexval.ival;
209     return(p);
210 }
211
212 Pnode newnode(Typenode tnode) {
213     Pnode p = malloc(sizeof(Node));
214     p->type = tnode;
215     p->p1 = p->p2 = p->p3 = NULL;
216     return(p);
217 }

```

```

217 void parser() {
218     yyin = stdin;
219     if(yyparse() == 0) {
220         treeprint(root, 0);
221     }
222     else {
223         yyerror();
224     }
225 }
226 }
227
228 void yyerror() {
229     fprintf(stderr, "Line %d: error on symbol \"%s\" \n", line, yytext);
230     exit(-1);
231 }
```

Figura 5.4: Il codice del file *parser.y*

Nelle dichiarazioni whitebox di Yacc vengono definiti i token di Tofu, che sono gli stessi restituiti dal lexer (vedi figura 5.2). La compilazione del *parser.y*, oltre che produrre un file con estensione .c, ha anche come output il file *parser.h* che associa un intero univoco a ciascun token e che viene incluso dal *lexer.lex* (figura 5.2). L'argomento della compilazione viene approfondito nel capitolo 7.

```

37 #ifndef YY_YY_PARSER_H_INCLUDED 66   LE = 272,
38 # define YY_YY_PARSER_H_INCLUDED 67   APP = 273,
39 /* Debug traces. */           68   NOT = 274,
40 #ifndef YYDEBUG               69   RETURN = 275,
41 # define YYDEBUG 0             70   INT = 276,
42 #endif                         71   STRING = 277,
43 #if YYDEBUG                   72   BOOL = 278,
44 extern int yydebug;           73   INTCONST = 279,
45 #endif                         74   STRCONST = 280,
46                           75   BOOLCONST = 281,
47 /* Token type. */            76   ID = 282,
48 #ifndef YYTOKENTYPE           77   ERROR = 283
49 # define YYTOKENTYPE          78 };
50 enum yytokentype              79 #endif
51 {                            80
52     VARIABLES = 258,          81 /* Value type. */
53     FUNCTIONS = 259,          82 #if ! defined YYSTYPE && ! defined YYSTYPE_IS_DEFINED
54     IF = 260,                  83     typedef int YYSTYPE;
55     THEN = 261,                84 # define YYSTYPE_IS_TRIVIAL 1
56     ELSE = 262,                85 # define YYSTYPE_IS_DEFINED 1
57     END = 263,                 86 #endif
58     OTHERWISE = 264,           87
59     BODY = 265,                88
60     SHOW = 266,                89     extern YYSTYPE yylval;
61     AND = 267,                 90
62     OR = 268,                  91     int yyparse (void);
63     EQ = 269,                  92
64     NE = 270,                  93 #endif /* !YY_YY_PARSER_H_INCLUDED */
65     GE = 271,                  94
```

Figura 5.5: Il codice del file *parser.h*

Le regole di traduzione esprimono la grammatica EBNF di Tofu (consultabile al-

l’Appendice B, insieme alla BNF) in cui ogni produzione ha associata una o più azioni semantiche che hanno il compito di costruire l’albero sintattico astratto. Dopo aver inizializzato la variabile **root** di tipo *Pnode* a NULL nelle dichiarazioni blackbox, si assegna a **root** il valore referenziato dalla pseudo-variabile \$\$, che inizialmente è l’assioma *program*, e si costruisce il nodo radice tramite la funzione *nontermnode(NPROGRAM)*. Dopodiché, tramite l’istruzione $\$\$ \rightarrow p1 = \1 , si collega il nodo radice al suo primo figlio, che in questo caso corrisponde a *var_section*; l’azione $\$1 \rightarrow p3 = \2 fa puntare il nonterminale *var_section* al fratello nonterminale *func_section* e $\$2 \rightarrow p3 = \3 collega *func_section* al fratello *body_section* (righe 17-19 della figura 5.4). Ogni regola di traduzione segue questa logica, rispettando i vincoli imposti dalla EBNF per la creazione dell’albero astratto. Analizzando la regola che va da linea 25 a 27, si possono notare alcuni aspetti interessanti:

- la produzione è ricorsiva dato che il nonterminale *var_decl_list* è presente sia a sinistra che a destra dei due punti;
- il nonterminale *var_decl_list* non è un nodo dell’albero astratto poiché gli viene assegnato il valore del figlio ($\$\$ = \1);
- la produzione alternativa a riga 26 *var_decl* svolge l’azione di default ($\$\$ = \1) come unica azione semantica e quindi si può omettere.

Ovunque compaia il terminale ID nelle regole di produzione, viene dichiarata l’azione embedded $\$\$ = idnode()$ e viene referenziata quest’ultima dalla pseudo-variabile. Ad esempio, a linea 33 è presente l’istruzione $\$\$ = \2 , dove \$2 riferisce l’azione embedded, anziché $\$\$ = \1 , dove \$1 riferenzia ID. Infine le azioni associate ai simboli terminali dell’albero, compreso ID, sfruttano funzioni dichiarate nell’ultima sezione di Yacc (righe 37-39, 123-125).

La funzione ausiliaria *newnode()* richiede un parametro di tipo *Typlenode* e crea un nuovo nodo allocando la memoria necessaria, assegnandogli il tipo ricevuto come parametro e inizializzando a NULL tutti i suoi puntatori. Tutte le successive funzioni costruiscono un nuovo nodo invocando la *newnode()*. La funzione *nontermnode* riceve in ingresso un *Nonterminal* e restituisce un nodo nonterminale, dopo averlo creato e aver impostato il suo valore intero con quello di *Nonterminal*. La *strconstnode()* crea il nodo terminale di tipo T_STRCONST e assegna al valore stringa del nodo il suo valore lessicale, reperito dal lexer. L'*idnode()*, l'*intconstnode()* e il *boolconstnode()* seguono lo stesso meccanismo. La *keynode()* ha un parametro di tipo *Typlenode* e costruisce il nodo terminale che dichiara la tipologia di ciascuna variabile in Tofu (T_INT, T_STRING, T_BOOL). Nella funzione *parser()* si dichiara di utilizzare lo standard input (stdin), cioè l'input da tastiera o un file in ingresso, e si invoca la *yyparse()*: se è tutto corretto, l'albero astratto viene costruito e stampato su file tramite la *treeprint()*; altrimenti si chiama la *yyerror()* che segnala un messaggio d'errore all'utente. La funzione *treeprint()* è dichiarata in un programma C a parte, denominato *tree.c*.

```

1  #include "def.h"          32   "HE",
2                               33   ">",
3  char* tabtypes[] =          34   "GE",
4  {                           35   "<",
5    "INT",                   36   "LE",
6    "STRING",                37   ".",
7    "BOOL",                  38   "_",
8    "INTCONST",              39   "''",
9    "BOOLCONST",             40   "''",
10   "STRCONST",              41   "APP",
11   "ID",                    42   ".",
12   "NONTERMINAL"           43   "NOT",
13 };                          44   "RANGE",
14                           45   "OPT_EXPR_LIST",
15 char* tabnonterm[] =        46   "IF_EXPR",
16 {                           47   "INPUT",
17   "PROGRAM",               48   "OUTPUT",
18   "VAR_SECTION",           49   "FUNC_CALL",
19   "VAR_DECL",              50   "BODY_SECTION",
20   "ID_LIST",               51   "STAT_LIST",
21   "TYPE",                  52   "STAT",
22   "LIST_TYPE",              53   "ASSIGN_STAT",
23   "FUNC_SECTION",          54   "SHOW_STAT"
24   "FUNC_DECL",             55 };
25   "OPT_FORMAL_LIST",       56
26   "FORMAL_DECL",           57 void treeConstruction(FILE *fp, Pnode root, int indent)
27   "GUARD_EXPR",            58 {
28   "GUARD_LAST",             59   int i;
29   "AND",                   60   Pnode p;
30   "OR",                    61   Pnode sc;
31   "EQ",                   62
32   "GE",                   63   for(i=0; i<indent; i++)
33   "<",                   64   fprintf(fp, "%s", "    ");
34   "LE",                   65   fprintf(fp, "%s", (root->type == T_NONTERMINAL ?
35   ".\"",                 66   tabnonterm[root->value.ival] :
36   "''",                  67   tabtypes[root->type]));
37   "APP",                  68   if(root->type == T_ID || root->type == T_STRCONST)
38   ".\"",                 69   fprintf(fp, " (%s)", root->value.sval);
39   "NOT",                  70   else if(root->type == T_INTCONST)
40   "''",                  71   fprintf(fp, " (%d)", root->value.ival);
41   "RANGE",                72   else if(root->type == T_BOOLCONST)
42   ".",                   73   fprintf(fp, " (%s)", (root->value.ival == 1 ?
43   "NOT",                 74   "true" : "false"));
44   "OPT_EXPR_LIST",         75   fprintf(fp, "%s", "\n");
45   "IF_EXPR",               76
46   "INPUT",                 77   for(p=root->p1; p != NULL; p = p->p3) {
47   "OUTPUT",                78   sc = root->p2;
48   "FUNC_CALL",             79   treeConstruction(fp, p, indent+1);
49   "BODY_SECTION",          80   if(sc != NULL)
50   "STAT_LIST",             81   | treeConstruction(fp, sc, indent+1);
51   "STAT",                  82 }
52   "ASSIGN_STAT",           83
53   "SHOW_STAT"              84
54
55 }                           85 void treeprint(Pnode root, int indent)
56
57 void treeConstruction(FILE *fp, Pnode root, int indent)
58 {
59   int i;
60   Pnode p;
61   Pnode sc;
62
63   for(i=0; i<indent; i++)
64   fprintf(fp, "%s", "    ");
65   fprintf(fp, "%s", (root->type == T_NONTERMINAL ?
66   tabnonterm[root->value.ival] :
67   tabtypes[root->type]));
68   if(root->type == T_ID || root->type == T_STRCONST)
69   fprintf(fp, " (%s)", root->value.sval);
70   else if(root->type == T_INTCONST)
71   fprintf(fp, " (%d)", root->value.ival);
72   else if(root->type == T_BOOLCONST)
73   fprintf(fp, " (%s)", (root->value.ival == 1 ?
74   "true" : "false"));
75   fprintf(fp, "%s", "\n");
76
77   for(p=root->p1; p != NULL; p = p->p3) {
78     sc = root->p2;
79     treeConstruction(fp, p, indent+1);
80     if(sc != NULL)
81     | treeConstruction(fp, sc, indent+1);
82   }
83
84
85 void treeprint(Pnode root, int indent)
86 {
87   FILE *fp;
88   fp = fopen("AbstractTree.txt", "w");
89
90   treeConstruction(fp, root, indent);
91
92   fclose(fp);
93 }
```

Figura 5.6: Il codice C del file *tree.c*

La figura sottostante illustra come appare la stampa dell'albero astratto del programma Tofu *inverti_liste.tofu* (vedi figura 4.4) sul file *AbstractTree.txt*.

```

1   PROGRAM
2     VAR_SECTION
3       VAR_DECL
4         ID_LIST
5           ID (listaInput)
6           ID (invertita)
7           TYPE
8             LIST_TYPE
9               TYPE
10              LIST_TYPE
11                TYPE
12                  STRING
13
14     VAR_DECL
15       ID_LIST
16         ID (msgInserimento)
17         TYPE
18             STRING
19
20   FUNC_SECTION
21     FUNC_DECL
22       ID (invertiSingolaLista)
23       OPT_FORMAL_LIST
24         FORMAL_DECL
25           ID (lista)
26           TYPE
27             LIST_TYPE
28               TYPE
29                 STRING
30
31     TYPE
32       LIST_TYPE
33         TYPE
34             STRING
35
36     GUARD_EXPR
37       FUNC_CALL
38         ID (empty)
39         OPT_EXPR_LIST
40           ID (lista)
41
42     OPT_EXPR_LIST
43
44     GUARD_LAST
45       APP
46         OPT_EXPR_LIST
47           FUNC_CALL
48             ID (last)
49             OPT_EXPR_LIST
50               ID (lista)
51
52     FUNC_CALL
53       ID (invertiSingolaLista)
54       OPT_EXPR_LIST
55         FUNC_CALL
56           ID (init)
57             OPT_EXPR_LIST
58               ID (lista)
59
60     FUNC_DECL
61       ID (inverti)
62       OPT_FORMAL_LIST
63         FORMAL_DECL
64           ID (daInvertire)
65           TYPE
66             LIST_TYPE
67               TYPE
68                 LIST_TYPE
69                   TYPE
70                     STRING

```

```

63      TYPE
64          LIST_TYPE
65              TYPE
66                  LIST_TYPE
67                      TYPE
68                          STRING
69      GUARD_EXPR
70          FUNC_CALL
71              ID (empty)
72          OPT_EXPR_LIST
73              ID (daInvertire)
74          OPT_EXPR_LIST
75      GUARD_LAST
76          APP
77              OPT_EXPR_LIST
78                  FUNC_CALL
79                      ID (invertiSingolaLista)
80                  OPT_EXPR_LIST
81                      FUNC_CALL
82                          ID (last)
83                  OPT_EXPR_LIST
84                      ID (daInvertire)
85          FUNC_CALL
86              ID (inverti)
87          OPT_EXPR_LIST
88              FUNC_CALL
89                  ID (init)
90          OPT_EXPR_LIST
91                  ID (daInvertire)
92      STAT_LIST
93          SHOW_STAT
94              STRCONST ("Questo programma inverte una lista
95                          di liste di stringhe.\n")
96          ASSIGN_STAT
97              ID (msgInserimento)
98              STRCONST ("Inserisci una lista di stringhe con
99                          profondità 2: ")
100         SHOW_STAT
101             ID (msgInserimento)
102         ASSIGN_STAT
103             ID (listaInput)
104             INPUT
105                 TYPE
106                     LIST_TYPE
107                         TYPE
108                             LIST_TYPE
109                                 TYPE
110                                     STRING
111         ASSIGN_STAT
112             ID (invertita)
113             FUNC_CALL
114                 ID (inverti)
115                 OPT_EXPR_LIST
116                     ID (listaInput)
117         SHOW_STAT
118             STRCONST ("Lista invertita = ")
119         SHOW_STAT
120             ID (invertita)

```

Figura 5.7: L'albero sintattico astratto del programma Tofu *inverti_liste.tofu*

Se invece nel programma *inverti_liste.tofu* venisse dichiarata una nuova variabile all'interno del **body**, commettendo un errore sintattico, la funzione *parser()* invocherebbe la *yyerror()* che produrrebbe il seguente output:

```
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./T_code <programs/inverti_liste.tofu
Line 15: error on symbol ":"
```

Figura 5.8: Esempio di errore sintattico in Tofu

5.3 L'analizzatore semantico

L'analizzatore semantico di Tofu è scritto in C, come il generatore di codice intermedio e la macchina virtuale; dal paragrafo 3.2.3, si evince che i suoi ruoli principali sono due: la costruzione della **Symbol Table** e il **type checking**. Prima di spiegare la struttura della Symbol Table, bisogna fare una premessa e dire come si è scelto di catalogare gli identificatori, che in Tofu rappresentano variabili (globali), nomi delle funzioni e i loro parametri (variabili locali). Questo avviene tramite una *funzione hash*, che data una chiave in ingresso (l'identificatore) la trasforma in un indice (un intero) generando una Hash Table. La Hash Table non è altro che un array che va da **0** a **TOT-1**, dove **TOT** è la dimensione dell'array, la quale viene impostata nel codice dell'analizzatore ed è preferibile che sia un numero primo nell'ordine delle centinaia/migliaia per una distribuzione migliore. Nello specifico si converte l'identificatore carattere per carattere, utilizzando il suo valore intero secondo lo standard ASCII; poi lo si somma col valore del carattere precedente (zero se si sta analizzando il primo carattere) moltiplicato con un certo “peso”, detta anche operazione di SHIFT. Infine il risultato viene modularizzato ad un intero presente nel range [0-TOT-1] e il processo si ripete finché non si arriva all'ultimo carattere dell'identificatore: è restituito un indice appartenente a [0-TOT-1]. Si è deciso di utilizzare una Hash Table per gli identificatori globali (variabili e funzioni), impostando $TOT = 1031$, e una per i

parametri, con TOT_FUNC = 191. In entrambe la costante SHIFT vale 4.

```
76  int hash (char* id, int dim) {  
77  |  int h=0;  
78  |  for(int i=0; id[i] != '\0'; i++)  
79  |  |  h = ((h << SHIFT) + id[i]) % dim;  
80  |  return h;  
81 }
```

Figura 5.9: Il codice C della funzione *hash()*

Per esempio, se dichiaro la variabile **x1** in Tofu, il suo indice sarà:

$$h = ((0 * (2^4 = 16)) + 120) \% 1031 = 120$$

$$h = ((120 * 16) + 49) \% 1031 = 1969 \% 1031 = 938$$

L'indice associato alla variabile **x1** è **938**

La Symbol Table è strettamente connessa alla Hash Table: infatti è un array della sua stessa dimensione, in cui ogni elemento corrisponde a una riga della Symbol Table. In aggiunta, la posizione della riga viene stabilita dal valore hash del nome dell'identificatore: la variabile **x1** è contenuta nella riga numero **938**. La riga della Symbol Table è una struttura articolata che, oltre al nome, contiene:

- La *classe* dell'identificatore, cioè se è una variabile, una funzione o un parametro (informazione rappresentata dall'*enum Class* nel codice).
- L'*oid* (object identifier) relativo all'identificatore: è un numero univoco che lo identifica all'interno del programma Tofu. Si segue una numerazione crescente per ogni classe: per le variabili, da 0 in poi, e per le funzioni, da 1 in poi, è una numerazione assoluta (dove gli oid da 1 a 6 sono riservati per le funzioni built-in); mentre per i parametri, da 0 in poi, è una numerazione locale. L'oid verrà sfruttato dal generatore di codice intermedio e dalla macchina virtuale.
- Il *tipo* dell'identificatore (*struct Type*) caratterizzato da un dominio che è INT, STRING, BOOL o LIST (*enum Domain*), da una profondità (depth) e da un

tipo atomico (atomic). La profondità indica il numero di livelli di una lista, mentre l'*enum Atomic* ne rappresenta il tipo atomico: INT, STRING, BOOL o UNKNOWN. Se Domain \neq LIST, il resto delle informazioni è superfluo: la profondità viene impostata a 0 e il tipo atomico in UNKNOWN. L'atomic UNKNOWN però, non serve solo a questo, ma viene usato anche in caso di lista vuota. Ad esempio Domain = LIST, depth = 3 e Atomic = BOOL, indica una lista di liste di liste di tipo atomico booleano.

- *Formals* che è un campo utilizzato solo dalle funzioni, in quanto è una struttura contenente informazioni sul numero dei parametri e un vettore di puntatori agli stessi (*struct Formals*).
- La *local*, anch'essa usata solo dalle funzioni, che è una Symbol Table locale in cui sono classificati i parametri.
- Infine *next* è un puntatore al descrittore successivo: cioè se due identificatori risultano avere lo stesso indice hash (collidono), anziché sovrascriversi, il nome già presente in quell'indice si collega al nuovo nome.

```

52  typedef struct {
53      int num;
54      struct stable *params;
55  } Formals;
56
57  typedef struct stable {
58      char *name;
59      Class classe;
60      int oid;
61      Type tipo;
62      struct stable *local;
63      Formals formals;
64      struct stable *next;
65
66  } SymbolTable;
67
68  SymbolTable symbolTable[TOT];

```

Figura 5.10: Dichiarazione della Symbol Table in C

La creazione della Symbol Table principalmente avviene per controllare che non ci siano variabili, funzioni o parametri con nomi duplicati e che nelle espressioni vengano usati identificatori esistenti. Per svolgere tale compito ci si serve della funzione *lookup()* che, richiamando la funzione *hash()*, ricava il numero di riga della Symbol Table e verifica se l'identificatore presente in quella riga o quello collegato ad esso tramite il puntatore *next* coincide col nome ricevuto in ingresso dalla *lookup()*: se sono uguali viene restituito TRUE, altrimenti FALSE.

```

273 int lookup(char *id, SymbolTable st[], int dim) {
274     int ok = FALSE;
275     int h = binarySearch(dim, hash(id, dim));
276     if(st[h].name != NULL) {
277         if(strcmp(st[h].name, id) == 0 || strcmp(st[h].next->name, id) == 0) {
278             ok = TRUE;
279         }
280     }
281     return ok;
282 }
```

Figura 5.11: Il codice C della funzione *lookup()*

Quindi durante la dichiarazione di variabili, funzioni o parametri la *lookup()* deve ritornare FALSE, garantendo l'assenza di doppioni, mentre quando sta controllando un'espressione deve restituire TRUE, assicurando che si stanno usando identificatori noti al programma Tofu. Se in una delle due casistiche viene restituito un booleano inatteso, si segnala la presenza di un errore.

La funzione che si occupa di costruire la Symbol Table ed effettuare tutti i controlli necessari è la *createAndCheckSymbolTable()* che, per non avere troppe responsabilità, distribuisce il lavoro a molte altre funzioni ausiliarie. Il ruolo primario della *createAndCheckSymbolTable()* è di scorrere dall'alto verso il basso l'intero albero astratto, ricevuto dal parser, e di richiamare funzioni che inseriscano i nodi terminali, escluse le costanti, con le loro informazioni annesse all'interno della Symbol Table, effettuando tutte le verifiche richieste. Si parte dalla sezione delle variabili, invocando

la *insertVariable()*, in seguito si passa alla sezione delle funzioni che, prima vengono tutte catalogate nella Symbol Table tramite la *insertFunction()*, ignorando le loro espressioni, e solo in un secondo momento si effettua il type checking sulle espressioni. Il motivo è poter richiamare una generica funzione **f**' all'interno dell'espressione di una funzione **f**, senza che **f**' sia stata ancora dichiarata. Una volta inserita una funzione, la *createAndCheckSymbolTable()* invoca la *insertParameter()* per catalogare ciascuno dei suoi parametri nella local Symbol Table. Infine controlla che nel corpo del programma non ci siano errori semanticici.

```

805 void insertVariable(Pnode pnode) {
806     for(Pnode p = pnode->p1; p != NULL; p = p->p3) {
807         Pnode var = p->p1->p1;
808         Pnode t = p->p1->p3;
809         for(var; var != NULL; var = var->p3) {
810             char *name = var->value.sval;
811             if(lookup(name, symbolTable, TOT) == FALSE) {
812                 Type tipo = defineType(t->p1);
813                 int index = binarySearch(TOT, hash(name, TOT));
814                 whereInsert(symbolTable, index, name, VAR, tipo);
815             }
816             else {
817                 fprintf(stderr, "Error: identifier \"%s\" is a duplicate variable \n", name);
818                 exit(-1);
819             }
820         }
821     }
822 }
```

Figura 5.12: Il codice C della funzione *insertVariable()*

Analizziamo nel dettaglio la *insertVariable()*. Il ciclo **for** più esterno scorre tutte le dichiarazioni: la variabile *p* punta al primo nonterminale di tipo VAR_DECL e poi viene aggiornata facendole riferire il fratello, che è ancora un nonterminale di tipo VAR_DECL, finché non ci sono più fratelli e dunque *p* vale NULL, uscendo così dal ciclo (riga 806 della figura 5.12). All'interno del **for**, il Pnode *var* viene inizializzato col terminale di tipo ID e il Pnode *t* con il nonterminale di tipo TYPE. Si esegue un ciclo interno sui fratelli di *var*, che in Tofu corrisponde alla dichiarazione di più variabili su un'unica linea dello stesso tipo, si estrae il suo valore lessicale, cioè il

nome dell’identificatore, e si richiama la *lookup()*. Se quest’ultima restituisce TRUE, allora viene stampato un messaggio d’errore e il programma termina (righe 817 e 818). Altrimenti tramite la funzione ausiliaria *defineType()* si recupera il tipo della variabile (dominio, profondità e tipo atomico), il suo indice con la *hash()* e si inserisce la variabile in questione nella Symbol Table. Tale operazione di fatto viene svolta dalla *whereInsert()* che richiede come parametri la Symbol Table (globale o locale), l’indice hash, il nome dell’identificatore, la sua classe (VAR, FUNC o PARAM) e il suo tipo. Le funzioni *insertFunction()* e *insertParameter()* agiscono in maniera analoga.

Informazioni importanti che devono essere memorizzate dalla Symbol Table sono le funzioni built-in di Tofu. Infatti, nonostante non vengano mai dichiarate esplicitamente in un programma Tofu, quando sono invocate devono essere riconosciute: per tale ragione l’analizzatore semantico le inserisce in automatico all’interno della Symbol Table. La funzione *insertBuiltInFunc()*, invocata subito dalla *createAndCheckSymbolTable()*, ha proprio questo scopo.

```

325 void builtinFuncStruct(char *name, Type tipo) {
326     Type param = {D_LIST, MENO_UNO, A_UNKNOWN};
327     int i = binarySearch(TOT, hash(name, TOT));
328     whereInsertFunc(symbolTable, i, name, FUNC, tipo, UNO);
329     int local_i = binarySearch(TOT_FUNC, hash(name, TOT_FUNC));
330     whereInsert(symbolTable[i].local, local_i, "param1", PARAM, param);
331     symbolTable[i].formals.params[0] = symbolTable[i].local[local_i];
332 }
333
334 void insertBuiltinFunc() {
335     Type e_tipo = {D_BOOL, 0, A_UNKNOWN};
336     builtinFuncStruct(EMPTY, e_tipo);
337
338     Type l_tipo = {D_INT, 0, A_UNKNOWN};
339     builtinFuncStruct(LENGTH, l_tipo);
340
341     Type htli_tipo = {D_LIST, UNO, A_UNKNOWN};
342     builtinFuncStruct(HEAD, htli_tipo);
343
344     builtinFuncStruct(TAIL, htli_tipo);
345
346     builtinFuncStruct(LAST, htli_tipo);
347
348     builtinFuncStruct(INIT, htli_tipo);
349 }

```

Figura 5.13: Il codice C delle funzioni *insertBuiltinFunc()* e *builtinFuncStruct()*

La *insertBuiltinFunc()* semplicemente definisce il tipo restituito da ciascuna funzione built-in, dopodiché richiama la *builtinFuncStruct()* passandogli in input il nome specifico della funzione (empty, length, head, tail , last o init) e il suo tipo. Si noti che per la head, la tail , la init e la last è stato definito il tipo in comune *htli_tipo*, che tuttavia verrà aggiornato dinamicamente durante la valutazione delle espressioni. In questo modo, la funzione *head()* applicata a una lista di stringhe può restituire una stringa, mentre se applicata a una lista con profondità 2 di interi può restituire una lista con profondità 1 di interi. La *builtinFuncStruct()* aggiunge la funzione built-in in ingresso, tramite la *whereInsertFunc()*, nella riga della Symbol Table ricavata con la *hash()* del suo nome. In seguito, dato che tutte le funzioni built-in richiedono un unico parametro, viene richiamata una sola volta la *whereInsert()* che lo inserisce all'interno della local Symbol Table. Del parametro in input, si sa con certezza che deve avere Domain = LIST, ma la sua profondità e il tipo atomico non si conosco-

no a priori: anche quest'ultimi saranno calcolati dinamicamente. Le valutazioni “in tempo reale” del parametro e del tipo restituito dalle funzioni built-in, sono svolte durante il loro type checking dalla funzione *checkBuiltInFunc()*, che entra in gioco quando una di esse è usata nel programma Tofu.

Il type checking effettivo è eseguito dalla funzione *exprType()* che richiede un parametro di tipo Pnode e che è invocata dalla *createAndCheckSymbolTable()* sia per accertarsi che il tipo restituito dalla funzione e quello della sua espressione coincidano, sia per verificare la correttezza delle espressioni presenti nelle istruzioni Tofu di assegnamento e stampa a video. La *exprType()* durante l’analisi di espressioni complesse si chiama ricorsivamente fino ad arrivare a un nodo terminale (variabile, parametro o costante che sia) e ne preleva il tipo. Allora comincia a risalire controllando la compatibilità di ciascuna espressione, fino a restituire il tipo associato all’intera espressione. Di seguito vengono riportati degli esempi di type checking su alcune delle operazioni ammesse in Tofu.

```

752     else if(expr->value.ival == NAND || expr->value.ival == NOR) {
753         return andOrExprType(expr, t1, t2);
754     }
755
483     Type andOrExprType(Pnode expr, Type t1, Type t2) {
484         t1 = exprType(expr->p1);
485         t2 = exprType(expr->p2);
486         if(t1.domain != D_BOOL) {
487             printf("%s", msg_error);
488             fprintf(stderr, "Error: expected a BOOL type and not %s type \n", describeDomain(t1.domain));
489             exit(-1);
490         }
491         if(t2.domain != D_BOOL) {
492             printf("%s", msg_error);
493             fprintf(stderr, "Error: expected a BOOL type and not %s type \n", describeDomain(t2.domain));
494             exit(-1);
495         }
496         return t1;
497     }

```

Figura 5.14: Un frammento di codice C della funzione *exprType()* e il codice della funzione *andOrExprType()*

Se il nonterminale corrisponde ad OR o AND, allora *exprType()* restituisce la

andOrExprType(), in cui si verifica, richiamando ricorsivamente la *exprType()*, che i due argomenti dell'operazione di **and** o di **or** siano entrambi booleani. In caso contrario si termina il programma mostrando un messaggio d'errore. Se invece risulta tutto nella norma, la *andOrExprType()* restituisce un tipo **bool**.

```

775     else if(expr->value.ival == NRANGE) {
776         return rangeExprType(expr, t1, t2);
777     }
778
779     Type rangeExprType(Pnode expr, Type t1, Type t2) {
780         t1 = exprType(expr->p1);
781         t2 = exprType(expr->p2);
782         if(t1.domain != D_INT) {
783             printf("%s", msg_error);
784             fprintf(stderr, "Error: expected a INT type and not %s type \n", describeDomain(t1.domain));
785             exit(-1);
786         }
787         if(t2.domain != D_INT) {
788             printf("%s", msg_error);
789             fprintf(stderr, "Error: expected a INT type and not %s type \n", describeDomain(t2.domain));
790             exit(-1);
791         }
792         t1.domain = D_LIST;
793         t1.depth = 1;
794         t1.atomic = A_INT;
795         return t1;
    }
```

Figura 5.14: Un frammento di codice C della funzione *exprType()* e il codice della funzione *rangeExprType()*

Il discorso è simile all'esempio precedente: se il nonterminale è di tipo RANGE, allora *exprType()* restituisce la *rangeExprType()*. Questa funzione controlla che gli operandi siano di tipo **int** e restituisce una lista di interi con profondità 1.

In conclusione la *exprType()* è formata da una cascata di if-else e, a seconda della tipologia del non terminale, invoca la funzione specifica per svolgere il type checking. Bisogna prestare particolare attenzione nell'effettuare i controlli riguardanti le liste, di cui è responsabile la funzione *optionalExprListExprType()*. Ogni elemento di una lista deve avere lo stesso tipo atomico e, in caso sia composta da altre liste, devono avere tutte uguale profondità. Tuttavia fanno eccezione le liste vuote, che possono essere inserite in una lista qualsiasi, a prescindere dalla sua depth e dal suo atomic.

Dunque UNKNOWN è un tipo atomico compatibile con tutti gli altri, che invece sono incompatibili fra loro. Esempi:

- list: [[int]] ... list = [[3, 4], [6, 7, 8], [**true**]]

Verrà segnalato un errore perché una lista che dovrebbe avere tipo atomico INT, presenta un elemento con atomic BOOL ([**true**])

- list: [[int]] ... list = [[[3, 4]], [6, 7, 8], [10]]

Verrà segnalato un errore perché una lista che dovrebbe avere profondità 2, contendo liste di depth 1, presenta un elemento con depth pari a 2 ([[3, 4]]).

- list: [[[int]]] ... list = [[[1, 3, 5], []], [], [{}], [[127]]]

L’assegnamento della variabile *list* è corretto.

In ultima istanza viene dichiarata la funzione *semantics()* che invoca la *parser()* di Yacc, la *createAndCheckSymbolTable()* e scorre tutti gli elementi non nulli presenti nella Symbol Table, stampandoli sul file *SymbolTable.txt*. Il codice C completo dell’analizzatore semantico (file *semantics.c*) è consultabile all’Appendice C.

```

989 void semantics() {
990     parser();
991     createAndCheckSymbolTable();
992
993     FILE *fp;
994     fp = fopen("SymbolTable.txt", "w");
995
996     for(int i=0; i < TOT; i++) {
997         if(symbolTable[i].name != NULL) {
998             fprintf(fp, "Contenuto Symbol Table nel posto %d \n", i);
999             printElement(fp, symbolTable[i], i, 0);
1000         }
1001     }
1002     fclose(fp);
1003 }
```

Figura 5.15: Il codice C della funzione *semantics()*

Nella figura 5.16 viene riportato il contenuto del file *SymbolTable.txt* per il programma *inverti_liste.tofu*.

```

1 Contenuto Symbol Table nel posto 51
2 Nome: last, Classe: FUNCTION, Oid:5, Numero_Parametri: 1
3
4 Contenuto Symbol Table nel posto 210
5 Nome: init, Classe: FUNCTION, Oid:6, Numero_Parametri: 1
6
7 Contenuto Symbol Table nel posto 272
8 Nome: inverti, Classe: FUNCTION, Tipo: (Dominio: LIST, Profondita: 2, Tipo_Atomico: STRING), Oid:8, Numero_Parametri: 1
9
10 Parametri della funzione:
11
12 Contenuto della Local Symbol Table nel posto 107
13 Nome: daInvertire, Classe: PARAMETER, Tipo: (Dominio: LIST, Profondita: 2, Tipo_Atomico: STRING), Oid:0
14
15 Contenuto Symbol Table nel posto 324
16 Nome: invertiSingolaLista, Classe: FUNCTION, Tipo: (Dominio: LIST, Profondita: 1, Tipo_Atomico: STRING), Oid:7, Numero_Parametri: 1
17
18 Parametri della funzione:
19
20 Contenuto della Local Symbol Table nel posto 18
21 Nome: lista, Classe: PARAMETER, Tipo: (Dominio: LIST, Profondita: 1, Tipo_Atomico: STRING), Oid:0
22
23 Contenuto Symbol Table nel posto 446
24 Nome: invertita, Classe: VARIABLE, Tipo: (Dominio: LIST, Profondita: 2, Tipo_Atomico: STRING), Oid:1
25
26 Contenuto Symbol Table nel posto 690
27 Nome: tail, Classe: FUNCTION, Oid:4, Numero_Parametri: 1
28
29 Contenuto Symbol Table nel posto 757
30 Nome: length, Classe: FUNCTION, Oid:2, Numero_Parametri: 1
31
32 Contenuto Symbol Table nel posto 800
33 Nome: listaInput, Classe: VARIABLE, Tipo: (Dominio: LIST, Profondita: 2, Tipo_Atomico: STRING), Oid:0
34
35 Contenuto Symbol Table nel posto 883
36 Nome: head, Classe: FUNCTION, Oid:3, Numero_Parametri: 1
37
38 Contenuto Symbol Table nel posto 907
39 Nome: empty, Classe: FUNCTION, Oid:1, Numero_Parametri: 1
40
41 Contenuto Symbol Table nel posto 957
42 Nome: msgInserimento, Classe: VARIABLE, Tipo: (Dominio: STRING, Profondita: 0, Tipo_Atomico: UNKNOWN), Oid:2

```

Figura 5.16: La stampa della Symbol Table del programma *inverti_liste.tofu*

Per chiudere il discorso riguardante l’analizzatore semantico, di seguito sono riportati alcuni errori che potrebbero verificarsi prendendo come riferimento il programma *inverti_liste.tofu*.

```
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./T_code <programs/inverti_liste.tofu
Error: identifier "listaInput" is a duplicate variable
```

Figura 5.17: Errore semantico: duplicazione della variabile **listaInput**

```
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./T_code <programs/inverti_liste.tofu
In the statement number 2:
Error: incompatible types; must be all STRING or INT
```

Figura 5.18: Errore semantico: assegnamento di un intero alla variabile stringa **msgInserimento**

```
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./T_code <programs/inverti_liste.tofu
In the statement number 4:
Error: incompatible list types; must both have depth 2 or 1
```

Figura 5.19: Errore semantico: assegnamento di una lista semplice di stringhe alla variabile **listaInput**, che è di tipo lista di stringhe con profondità 2.

5.4 Il codice intermedio T-code

Il codice intermedio di Tofu prende il nome di **T-code** e ha una struttura simile a quella del P-code (codice intermedio del Pascal). Prima si analizzeranno gli schemi di traduzione adottati, dopodiché si spiegherà nel dettaglio l'implementazione del generatore di T-code.

5.4.1 Gli schemi di traduzione

Schema generale (con riferimento alla figura 4.4):

```
TCODE size
VARS 3
⟨body⟩
HALT
⟨inverti⟩
⟨invertiSingolaLista⟩
```

dove *size* è il numero totale di istruzioni (senza contare se stessa), la VARS dichiara il numero di variabili presenti e la HALT l'istruzione di terminazione. Ciò che è racchiuso fra ⟨ ⟩ indica il codice intermedio del **body** e delle singole funzioni del programma.

Dichiarazione di variabili:

| | |
|------------|------|
| a: int; | NEWI |
| b: bool; | NEWB |
| s: string; | NEWS |
| l: [bool]; | NEWL |

Referenza a costante atomica:

| | | |
|------------------|---------------|--------------------|
| nat = 18; | LOCI 18 | Nota: true vale 1, |
| ok = false; | LOCB 0 | false 0 |
| nome = "Angela"; | LOCS "Angela" | |

Costruttore di una lista:

| | | |
|---------------------|---------|------------------------|
| numeri: [int]; | LOCI 25 | LIST è l'istruzione di |
| numeri = [25,9,67]; | LOCI 9 | impacchettamento e ha |
| | LOCI 67 | come argomento la |
| | LIST 3 | lunghezza della lista |

Referenza a un identificatore:

| | | |
|-------|------------|-----------------|
| j + 2 | LOAD 0 ^ j | Oggetto globale |
| not i | LOAD 1 ^ i | Oggetto locale |

Gli argomenti di LOAD sono il *flag* (0 se globale, 1 se locale) e l'*oid* (object identifier) che viene recuperato dalla Symbol Table. \wedge *name* indica l'oid dell'oggetto di nome *name* (quindi, un intero: 0, 1, 2, ...).

Assegnamento:

| | | |
|--------------|--------|--------------------|
| check: bool; | LOCB 1 | STOR ha l'oid come |
|--------------|--------|--------------------|

check = true; STOR ^ check unico argomento

Operazioni di negazione **-**, **not**:

| | | |
|-------------------------------|-------------------------------|---|
| $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ rappresenta il T-code |
| UMIN | NEGA | di una qualsiasi espressione |

Operazioni di aritmetiche **+**, **-**, *****, **/**:

| | | | |
|-------------------------------|-------------------------------|-------------------------------|-------------------------------|
| $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ |
| PLUS | MINU | MULT | DIVI |

Operazioni di confronto **==**, **!=**, **>**, **>=**, **<**, **<=**:

| | | |
|-------------------------------|-------------------------------|-------------------------------|
| $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ |
| EQUA | NEQU | GRTR |
| $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ |
| GREQ | LETH | LEEQ |

Operazione di **range**:

| | |
|-----------------------|----------------|
| start,end: int; | LOAD 0 ^ start |
| range: [int]; | LOAD 0 ^ end |
| ... | RANG |
| range = [start..end]; | STOR ^ range; |

Operazioni logiche:

| | |
|---------------------------------|---|
| and | SKPF è un salto condizionato a false (LOCB 0), |
| $\langle \text{expr}_1 \rangle$ | cioè $\langle \text{expr}_1 \rangle$ è falsa. $\text{offset} = \langle \text{expr}_2 \rangle + 2$ ed è il |

| | |
|--------------------------|--|
| SKPF <i>offset</i> | numero di istruzioni da saltare per andare a LOCB 0. |
| $\langle expr_2 \rangle$ | Altrimenti si valuta $\langle expr_2 \rangle$ e se ne restituisce il valore. SKIP 2 è un salto incondizionato che non valuta la LOCB 0 e va all'istruzione successiva. |
| SKIP 2 | |
| LOCB 0 | |
| or | Si valuta la $\langle expr_1 \rangle$: se è falsa si passa a valutare la $\langle expr_2 \rangle$, grazie a SKPF 3, e viene restituito il suo valore. Invece se $\langle expr_1 \rangle$ è vera, si restituisce true (LOCB 1) e tramite la SKIP <i>exit</i> si salta l' $\langle expr_2 \rangle$. <i>exit</i> = $ \langle expr_2 \rangle + 1$ |
| $\langle expr_1 \rangle$ | |
| SKPF 3 | |
| LOCB 1 | |
| SKIP <i>exit</i> | |
| $\langle expr_2 \rangle$ | |

La funzione built-in **append**:

| | |
|----------------------------|------------------|
| pari, dispari, app: [int]; | LOAD 0 ^ pari |
| ... | LOAD 0 ^ dispari |
| app = pari ++ dispari; | APPN |
| | STOR ^ app |

Le funzioni built-in **empty**, **length**, **head**, **tail**, **init**, **last**:

| | | |
|------------------------|------------------------|------------------------|
| $\langle expr \rangle$ | $\langle expr \rangle$ | $\langle expr \rangle$ |
| EMPT | LENG | HEAD |
| $\langle expr \rangle$ | $\langle expr \rangle$ | $\langle expr \rangle$ |
| TAIL | INIT | LAST |

L'espressione condizionale **if-else**:

| | |
|--------------------------|--|
| $\langle expr_1 \rangle$ | É lo stesso meccanismo usato per le operazioni |
|--------------------------|--|

| | |
|--------------------------|--|
| SKPF $offset$ | logiche. Se $\langle expr_1 \rangle$ è vera, si valuta la $\langle expr_2 \rangle$, |
| $\langle expr_2 \rangle$ | dopodiché si esce. Altrimenti si salta la $\langle expr_2 \rangle$ |
| SKIP $exit$ | e si valuta la $\langle expr_3 \rangle$. $offset = \langle expr_2 \rangle + 2$, |
| $\langle expr_3 \rangle$ | $exit = \langle expr_3 \rangle + 1$. |

Le **guardie**:

| | |
|------------------------------|---|
| $\langle cond_1 \rangle$ | $\langle cond_i \rangle$ è un'espressione booleana. Se $\langle cond_1 \rangle$ è |
| SKIPF $offset_1$ | vera, si valuta la $\langle expr_1 \rangle$ e si esce (cioè |
| $\langle expr_1 \rangle$ | si va all'istruzione successiva di $\langle expr_{n+1} \rangle$). |
| SKIP $exit_1$ | Altrimenti si passa a verificare la $\langle cond_2 \rangle$ e si |
| $\langle cond_2 \rangle$ | ripete quanto fatto sopra, fino ad arrivare a |
| SKIPF $offset_2$ | $\langle cond_n \rangle$. Se è vera si restituisce la $\langle expr_n \rangle$, |
| $\langle expr_2 \rangle$ | mentre se risulta falsa si valuta la $\langle expr_{n+1} \rangle$ |
| SKIP $exit_2$ | (espressione della clausola finale otherwise). |
| ... | $offset_i = \langle expr_i \rangle + 2$ |
| $\langle cond_n \rangle$ | $exit_i = 2(n - i) + \langle expr_{n+1} \rangle + 1 +$ |
| SKIPF $offset_n$ | $+ \sum_{j=i+1}^n \langle cond_j \rangle + \langle expr_j \rangle $ |
| $\langle expr_n \rangle$ | |
| SKIP $exit_n$ | |
| $\langle expr_{n+1} \rangle$ | |

Richiamo di una funzione all'interno del **body**:

| | | |
|-----------------|-------------|------------------------------|
| n1,n2,s: int; | LOAD 0 ^ n1 | PUSH ha come argomento il |
| ... | LOAD 0 ^ n2 | numero di parametri della |
| s = sum(n1,n2); | PUSH 2 | funzione. JUMP richiede in |
| | JUMP &sum | ingresso l'indirizzo,cioè il |

| | |
|------|---|
| APOP | numero, della prima istruzione |
| | del corpo della funzione all'interno del T-code |

Dichiarazione di una funzione:

| | | |
|-----------------------|------------|----------------|
| ... | FUNC ^ sum | FUNC ha come |
| sum(a:int,b:int)→ int | LOAD 1 ^ a | argomento |
| a + b; | LOAD 1 ^ b | l'oid locale |
| ... | PLUS | della funzione |
| | RETN | (1,2,ecc...) |

L'istruzione di input ?:

| | |
|-----------------------|---------------|
| num: int; | GETI |
| ok: bool; | STOR ^ num |
| colore: string | GETB |
| corsi:[string]] | STOR ^ ok |
| ... | GETS |
| num = ?(int); | STOR ^ colore |
| ok = ?(bool); | GETL 2 STRING |
| colore = ?(string); | STOR ^ corsi |
| corsi = ?([[string]]) | |

GETL richiede la profondità della lista e il suo tipo atomico come argomenti. Il valore del tipo **bool** viene letto come **true\false** da tastiera e memorizzato come intero **1\0**.

Le istruzioni di output ! e **show**:

| | |
|-------------------------------|-------------------------------|
| $\langle \text{expr} \rangle$ | $\langle \text{expr} \rangle$ |
| PUTS | SHOW |

5.4.2 Il generatore di T-code

Prima di tutto bisogna definire le strutture dati per la generazione del T-code.

La struttura *Stat* è composta da

- **Address:** un intero che indica il numero dell’istruzione all’interno del T-code.
- **Operator:** l’operatore dell’istruzione (LOAD, STOR, ecc...). È un enum dichiarato nel file *def.h* (vedi figura 5.1).
- **Args:** gli argomenti richiesti dall’istruzione T-code. Da nessuno fino ad un massimo di due e possono essere di tipo intero o stringa.
- **Next:** puntatore alla struttura *Stat* successiva.

La *Stat* (statement) rappresenta un’istruzione del codice intermedio.

La struttura *Code* è costituita da

- **Head:** la *Stat* iniziale del codice intermedio.
- **Size:** un intero che rappresenta la dimensione totale del T-code.
- **Tail:** l’ultima *Stat* del codice intermedio.

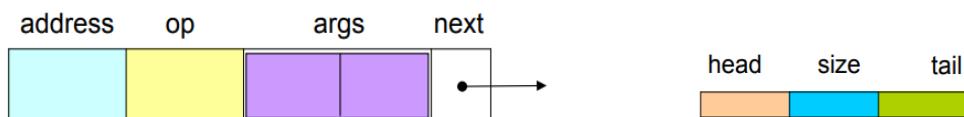


Figura 5.20: Le strutture Stat e Code

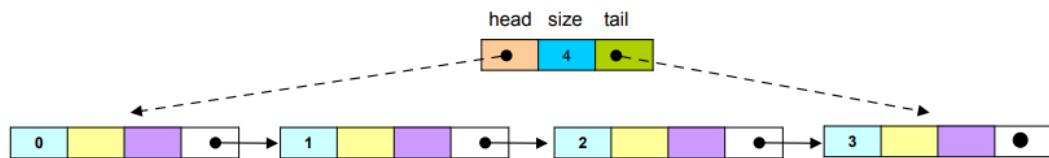


Figura 5.21: Rappresentazione di un segmento di codice (lista di istruzioni T-code)

Per costruire il T-code in questo modo, vengono dichiarate le seguenti funzioni:

- Stat **newstat*(Opeartor op): inizializza un nuovo *statement*;
- void *relocate*(Code code, int offset): aggiorna l'indirizzo di ogni istruzione presente nel parametro *code* tramite l'*offset*;
- Code *appcode*(Code code1, Code code2): concatena due frammenti di codice servendosi della *relocate()*;
- Code *concode*(Code code1, Code code2, ...): concatena tre o più frammenti di codice invocando la *appcode()*;
- *makecode*(Operator op): crea un'istruzione T-code senza argomenti richiamando la *newstat()*;
- Code *makecode1*(Operator op, int arg): crea un'istruzione T-code con un argomento invocando la *newstat()*;
- Code *makecode2*(Operator op, int arg1, int arg2): crea un'istruzione T-code che necessita di due argomenti richiamando la *newstat()*;
- Code *endcode*(): crea il frammento di codice terminale

Code code = {NULL, 0, NULL};

```

64  Stat *newstat(Operator op) {
65      Stat *pstat;
66      pstat = (Stat*) malloc(sizeof(Stat));
67      pstat->address = 0;
68      pstat->op = op;
69      pstat->next = NULL;
70      return pstat;
71  }
38  Code appcode(Code code1, Code code2) {
39      Code rescode;
40
41      relocate(code2, code1.size);
42      rescode.head = code1.head;
43      rescode.tail = code2.tail;
44      code1.tail->next = code2.head;
45      rescode.size = code1.size + code2.size;
46
47  }

```

Figura 5.22: Il codice C delle funzioni *newstat()* e *appcode()*

```

73  Code makecode(Operator op) {
74      Code code;
75      code.head = code.tail = newstat(op);
76      code.size = UNO;
77      return code;
78  }
80  Code makecode1(Operator op, int arg) {
81      Code code;
82      code = makecode(op);
83      code.head->args[0].ival = arg;
84      return code;
85  }

```

Figura 5.23: Il codice C delle funzioni *makecode()* e *makecode1()*

Per capire quale istruzione del codice intermedio creare, il generatore analizza l'albero sintattico prodotto dal *parser*. Tale compito viene affidato alla *genTcode()*, dato che si occupa della generazione vera e propria del codice intermedio. Più precisamente, essa richiama le opportune funzioni ausiliarie per la costruzione del codice. A loro volta queste ne invocheranno altre e si procede in questa maniera fino ad arrivare alle funzioni atomiche sopra viste. Al termine del processo descritto, la *genTcode()* fa un lavoro di rifinitura per avere a disposizione il T-code completo del programma Tofu.

```

779 void genTcode(FILE *fp) {
780     Code code = varBodyTcode();
781     Code func_code;
782     Pnode p = root->p1->p3->p1;
783     Pnode pSenzaJump = p;
784     verify_var = FALSE;
785
786     func_code = functcodeSenzaJump(pSenzaJump, code);
787     if(p != NULL) {
788         char *name = p->p1->value.sval;
789         func_index = binarySearch(TOT, hash(name, TOT));
790         Pnode expr = p->p1->p3->p3;
791         func_code = make_func_decl(symbolTable[func_index].oid, genExprCode(expr));
792         for(p = p->p3; p != NULL; p = p->p3) {
793             name = p->p1->value.sval;
794             func_index = binarySearch(TOT, hash(name, TOT));
795             expr = p->p1->p3->p3;
796             func_code = concode(func_code, make_func_decl(symbolTable[func_index].oid,
797                                         | genExprCode(expr)), endcode());
798         }
799     }
800     else {
801         func_code = endcode();
802     }

```

```

803     var_oid = 0;
804     if(func_code.size != 0) {
805         code = concode(varBodyTcode(), func_code, endcode());
806     }
807     else {
808         code = varBodyTcode();
809     }
810     fprintf(fp, "TCODE %d\n", code.size);
811     for(Stat *stat = code.head; stat != NULL; stat = stat->next) {
812         TcodeString(fp, stat->op, stat->args[0], stat->args[1]);
813     }
814 }
```

Figura 5.24: Il codice C della funzione *genTcode()*

```

246     Code make_func_decl(int fid, Code code) {
247         return concode(makecode1(T_FUNC, fid), code, makecode(RETN), endcode());
248     }
```

Figura 5.25: Il codice C della funzione *make_func_decl()*

La *genTcode()* invoca la funzione *varBodyTcode()* che crea un codice intermedio fittizio delle sezioni **variables** e **body** (cioè dall’istruzione VARS fino alla HALT). È fittizio in quanto all’istruzione JUMP viene passato come argomento un indirizzo errato, visto che le funzioni non sono ancora state dichiarate e dunque non si sa a che istruzione saltare. Dopodiché, sfruttando la variabile esterna *root*, si inizializza il puntatore al nodo col nonterminale FUNC DECL, cioè alla prima dichiarazione di una funzione: infatti a *genTcode()* interessa solo la parte dell’albero riguardante la sezione **functions**. La variabile *verify_var* è impostata a **false** perché si è interessati ai parametri (variabili locali), mentre è **true** nella *varBodyTcode()*, che riguarda le variabili globali. Tale controllo viene usato nella *genSimpleExprCode()* per decidere se cercare l’identificatore nella Symbol Table globale o in quella locale. Poi viene richiamata la *funcTcodeSenzaJump()*, che genera anch’essa un codice fittizio della sezione **functions** con lo scopo di calcolare la posizione in cui comincia la dichiarazione di ogni funzione e memorizzarla nell’array *entry[]*. Infatti, una volta svolto il suo ruolo, la *funcTcodeSenzaJump()* ritorna la *endcode()*. Il codice dalla riga 787

a 802 si occupa di creare il vero T-code della sezione **functions**. In particolare, il codice di ogni istruzione è costruito concatenando il codice esistente (*func_code*) con la *make_func_decl()* (figura 5.25) e con la *endcode()*. La *make_func_decl()* in ingresso vuole l'**oid** della funzione in questione e il T-code del suo corpo, prodotto dalla *genExprCode()*. Ora si può procedere alla reale generazione del T-code completo del programma Tofu, invocando nuovamente la *varBodyTcode()*, che questa volta assegna il giusto indirizzo alla JUMP, tramite la funzione *genFuncCallExprCode()*, che lo preleva dall'array *entry[]*. Infine, attraverso la *TcodeString()*, scrive l'intero codice intermedio sul file *fp* che la *genTcode()* richiede come parametro.

La funzione `genExprCode()` si comporta come la `exprType()` utilizzata dall’analizzatore semantico (vedi sezione 5.3), con la differenza che, anziché svolgere type checking, si occupa di generare il T-code specifico dei terminali e dei nonterminali collegati all’astrazione EXPR dell’albero sintattico. Esempio:

```
561     else if(expr->value.ival == NNOT) {
562         return concode(genExprCode(expr->p1), make_nega(), endcode());
563     }
564     else if(expr->value.ival == NRANGE) {
565         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_rang(), endcode());
566     }
567     else if(expr->value.ival == NIF_EXPR) {
568         return genIfExprCode(expr);
569     }
570 }
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1196
1197
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1296
1297
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1396
1397
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1489
1490
1491
1492
1493
1494
1495
1496
1497
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1596
1597
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1689
1690
1691
1692
1693
1694
1695
1696
1697
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2288
2289
2290
2291
2292
2293
2294
2295
2296
2296
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2396
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2488
2489
2490
2491
2492
2493
2494
2495
2496
2496
2497
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2588
2589
2590
2591
2592
2593
2594
2595
2596
2596
2597
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2688
2689
2690
2691
2692
2693
2694
2695
2696
2696
2697
2698
2699
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2788
2789
2790
2791
2792
2793
2794
2795
2796
2796
2797
2798
2799
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
287
```

```

417     Code genIfExprCode(Pnode expr) {
418         Code code_expr2 = genExprCode(expr->p1->p3);
419         Code code_expr3 = genExprCode(expr->p1->p3->p3);
420         return concode(genExprCode(expr->p1), make_if_expr(code_expr2.size + 2, code_expr2,
421             code_expr3.size + 1), code_expr3, endcode()));
422     }

```

Figura 5.26: Un frammento di codice C della funzione *genExprCode()* e il codice completo delle funzioni *make_nega()*, *make_rang()*, *make_if_expr()*, *genIfExprCode()*

Dalla figura 5.26 si evince che la *genExprCode()* effettua il controllo sulla corrispondenza tra il valore intero memorizzato dall'espressione e l'enum che rappresenta il nonterminale (NNOT, NRANGE, NIF_EXPR). Se, ad esempio, coincide con NRANGE, allora si restituisce la concatenazione dei frammenti di codice della *expr₁*, *expr₂* e della *make_rang()*. *expr₁*, *expr₂* sono le espressioni in input per l'operazione di **rang**e e il loro T-code viene generato invocando ricorsivamente la *genExprCode()*. La *make_rang()* semplicemente richiama la *makecode()* passandole come parametro l'operatore RANG. Il codice intermedio dell'operazione booleana **not** viene computato nello stesso modo. La funzione *genIfExprCode()*, invocata nella *genExprCode()*, la richiama a sua volta per calcolare il T-code delle *expr₁*, *expr₂* ed *expr₃* in ingresso. Infine restituisce la concatenazione del codice di *expr₁*, *make_if_expr()* ed *expr₃*. Si noti che l'argomento *offset* dell'istruzione SKPF vale code_expr2.size + 2 e quello *exit* della SKIP è code_expr3.size + 1. La logica appena spiegata è seguita da tutti gli altri nodi nonterminali e terminali (identificatori e costanti).

```

816     int main() {
817         semantics();
818
819         FILE *fp;
820         fp = fopen("IntermediateCode.txt", "w");
821         genTcode(fp);
822         fclose(fp);
823
824         return 0;
825     }

```

Figura 5.27: Il codice C del *main()*

Il generatore di codice intermedio è un programma C che deve essere eseguito, quindi, a differenza dell’analizzatore semantico, è dotato del *main()*. Esso invoca la funzione *semantics()*, vista alla sezione 5.3. Inoltre si occupa anche di creare o sovrascrivere un file di nome “*IntermediateCode*”, passato in input alla *genTcode()*, e in cui viene stampato il T-code completo: dunque rappresenta l’output del generatore. Il codice C completo del generatore di T-code (file *gen.c*) è consultabile all’Appendice D. Di seguito viene mostrato un esempio di codice intermedio di un programma Tofu.

```

1  TCODE 57
2  VARS 3
3  NEWL
4  NEWL
5  NEWS
6  LOCS "Questo programma inverte una lista di liste di stringhe.\n"
7  SHOW
8  LOCS "Inserisci una lista di stringhe con profondità 2: "
9  STOR 2
10 LOAD 0 2
11 SHOW
12 GETL 2 STRING
13 STOR 0
14 LOAD 0 0
15 PUSH 1
16 JUMP 38
17 APOP
18 STOR 1
19 LOCS "Lista invertita = "
20 SHOW
21 LOAD 0 1
22 SHOW
23 HALT
24 FUNC 7
25 LOAD 1 0
26 EMPT
27 SKPF 3
28 LIST 0
29 SKIP 10
30 LOAD 1 0
31 LAST
32 LIST 1
33 LOAD 1 0
34 INIT
35 PUSH 1
36 JUMP 22
37 APOP
38 APPN
39 RETN
40 FUNC 8
41 LOAD 1 0
42 EMPT
43 SKPF 3
44 LIST 0
45 SKIP 13
46 LOAD 1 0
47 LAST
48 PUSH 1
49 JUMP 22
50 APOP
51 LIST 1
52 LOAD 1 0
53 INIT
54 PUSH 1
55 JUMP 38
56 APOP
57 APPN
58 RETN

```

Figura 5.28: Il T-code del programma *inverti_liste.tofu*

6 La macchina virtuale di Tofu

L'architettura della macchina virtuale, detta anche T-machine o interprete (del T-code), è composta da una serie di stack (array), ognuno con i propri indici (puntatori impliciti). Essi rappresentano la memoria del codice, le variabili, le chiamate di funzioni, i parametri, gli oggetti temporanei e gli elementi di una lista.

La memoria del codice intermedio (*prog*) è una struttura formata da:

- **Operator:** l'operatore del T-code (SKIP, JUMP, ecc...);
- **args:** un array di massimo due elementi, che possono essere di tipo intero o stringa.

Il suo indice di riferimento è denominato *pc* (program counter).

L'activation stack relativo alle chiamate di funzioni (*astack*), con indice *ap*, è caratterizzato da tre valori interi:

- **num:** il numero di parametri della funzione;
- **objs:** l'indice dell'object stack (*op*) in cui avviene il caricamento del primo parametro della funzione;
- **ret:** l'indirizzo (*pc*) dell'istruzione T-code a cui ritornare una volta terminata l'invocazione.

Le variabili (array *vars* con indice *vp*), i parametri (stack *pstack* con indice *pp*), gli oggetti temporanei (stack *ostack* con indice *op*) e gli elementi di una lista (stack *istack* con indice *ip*) si basano sulla medesima struttura denominata *Object*:

- **Domain:** il tipo di un identificatore (INT, STRING, BOOL, LIST);

- **leng**: numero degli elementi di una lista (impostato a zero per i tipi atomici);
- **inst**: l’istanza dell’identificatore, che è intera per i tipi INT e BOOL ed è una stringa per STRING; invece per LIST questo valore è impostato sull’indice dell’*istack* (*ip*) in cui è presente il suo primo elemento.

In sostanza, il ruolo principale della T-machine è la manipolazione degli array sopra dichiarati. Ciò avviene caricando e prelevando gli oggetti del programma Tofu dagli stack, tramite l’incremento e il decremento dei rispettivi indici. Inoltre, quando necessario, vengono aggiornati i campi di *Object* relativi a ciascun oggetto. Le funzioni *push_obj()*, *push_int()*, *push_bool()*, *push_string()*, *push_list()*, *pop_obj()*, *pop_list()* agiscono sull’object stack.

```

81 void push_obj(Object obj) {
82     if(op >= MAX_OBJECTS)
83         error("Object stack overflow \n");
84
85     ostack[op++] = obj;
86 }
87
88 Object pop_obj() {
89     return ostack[--op];
90 }
```

```

91
92 void push_int(int n) {
93     Object obj;
94
95     obj.type = D_INT;
96     obj.leng = 0;
97     obj.inst.ival = n;
98     push_obj(obj);
99 }
```

Figura 6.1: Il codice C delle funzioni *push_obj()*, *pop_obj()* e *push_int()*

Le funzioni *exec_vars()*, *exec_newi()*, *exec_newb()*, *exec_news()*, *exec_newl()*, *exec_load()* ed *exec_stor()* agiscono sull’array *vars*.

```

165
166 void exec_vars(int num_vars) {
167     vars = malloc(num_vars*sizeof(Object));
168     size = num_vars;
169 }
170
171 void exec_newb() {
172     vars[vp].type = D_BOOL;
173     vars[vp].leng = 0;
174     vars[vp].inst.ival = -1;
175     vp++;
176 }
177
```

```

199 void exec_load(int flag, int oid) {
200     if(flag == 0) {
201         push_obj(vars[oid]);
202     }
203     else {
204         int current_param = pp - astack[ap].num;
205         push_obj(pstack[current_param + oid]);
206     }
207 }
208
209 void exec_stor(int oid) {
210     vars[oid] = pop_obj();
211 }
```

Figura 6.2: Il codice C delle funzioni *exec_vars()*, *exec_newb()*, *exec_load()* ed *exec_stor()*

L’*exec_vars()* alloca, tramite il parametro *num_vars*, lo spazio necessario affinché l’array *vars* possa contenere tutte le variabili del programma. La *exec_newb()* inserisce un booleano all’interno di *vars* in posizione *vp*, che viene incrementato alla fine. Si noti che l’istanza salvata durante la dichiarazione è un valore privo di senso: essa verrà sovrascritta quando si eseguirà un’operazione di assegnamento della variabile (*exec_stor()*). La *exec_load()* esegue azioni distinte a seconda che si tratti di una variabile o di un parametro: nel primo caso, carica l’oggetto sull’*ostack* prelevandolo dall’array *vars*; nel secondo caso lo carica estraendolo dal *pstack*.

Le funzioni *exec_push()*, *exec_apop()*, *exec_load()* lavorano sull’activation stack e sul parameters stack, mentre la *exec_jump()* e la *exec_retn()* solo sull’*astack*.

```

481 void exec_push(int params) {
482     astack[ap].num = params;
483     astack[ap].objs = op - astack[ap].num;
484     for(int i=0; i < params; i++)
485         pstack[pp++] = ostack[astack[ap].objs + i];
486     op = astack[ap].objs;
487 }
488 }
```

```

490 void exec_jump(int address) {
491     astack[ap].ret = pc;
492     pc = address;
493 }
```

Figura 6.3: Il codice C delle funzioni *exec_push()* ed *exec_jump()*

L’*exec_push()* assegna i campi **num** e **objs** di *astack* e carica tutti i parametri all’interno del *pstack*; infine aggiorna *op* col valore di **objs**. L’*exec_jump()* assegna il valore dell’indirizzo attuale (*pc*) al campo **ret** dell’activation stack, in cui sarà sempre presente l’operatore APOP. Poi modifica il *pc* con il suo argomento (*address*), in modo da saltare alla dichiarazione della funzione.

Le modifiche riguardanti l’instance stack sono meno immediate da individuare, dato che sono sparse all’interno della macchina virtuale: in generale qualunque funzione lavori con delle liste è autorizzata ad aggiornare l’*istack* e il rispettivo indice *ip*. Nella figura sottostante viene riportato l’esempio delle funzioni built-in *last* e *init*.

```

454 void exec_last() {
455     Object obj, last;
456     obj = pop_list();
457     if(obj.leng==0)
458         error("Function last applied to empty list \n");
459
460     last = istack[obj.leng + obj.inst.ival - 1];
461     push_obj(last);
462
463     ip = obj.leng + obj.inst.ival;
464     checkip();
465 }
466
467 void exec_init() {
468     Object obj, init;
469     obj = pop_list();
470     if(obj.leng==0)
471         error("Function init applied to empty list \n");
472
473     init = obj;
474     init.leng = obj.leng - 1;
475     push_obj(init);
476
477     ip = obj.leng + obj.inst.ival;
478     checkip();
479 }

```

Figura 6.4: Il codice C delle funzioni *exec_last()* ed *exec_init()*

L'*exec_last()* preleva la lista in cima all'object stack e controlla che non sia vuota: se lo è, viene segnalato un errore run-time e il programma termina. Anche la *init*, la *head* e la *tail* sono funzioni non definite su una lista vuota. Se la lista risulta avere lunghezza diversa da zero, si estrae il suo ultimo elemento dall'*istack* e lo si carica sull'*ostack*. Infine si aggiorna l'*ip*. Lo stesso meccanismo segue la *exec_init()*, che computa la lista privata del suo ultimo elemento, caratterizzata da una lunghezza decrementata di uno rispetto a quella della lista originale.

Ora si analizzano due esempi di come vengono manipolati gli stack durante l'esecuzione parziale di un programma Tofu.

Esempio 1

a, b: int;

TCODE *size_code*

| | | |
|--|---------|--------------|
| animali: [[string]]; | VARS 3 | LOCS “gatto” |
| ... | NEWI | LOCS “cane” |
| a = 20; | NEWI | LIST 2 |
| b = 39; | NEWL | LOCS “oca” |
| animali = [[“gatto”, “cane”],[“oca”]]; | LOCI 20 | LIST 1 |
| | STOR 0 | STOR 2 |
| | LOCI 39 | ... |
| | STOR 1 | |

Lo stack delle variabili (*vars*) risulta il seguente:

| vars | | | |
|------|--------|------|------|
| | DOMAIN | LENG | INST |
| 0 | INT | 0 | 20 |
| 1 | INT | 0 | 39 |
| 2 | LIST | 2 | 3 |

dove l’istanza della lista *animali* si riferisce alla posizione del suo primo elemento nell’*istack*, che è mostrato in questa tabella:

| istack | | | |
|--------|--------|------|---------|
| | DOMAIN | LENG | INST |
| 0 | STRING | 0 | “gatto” |
| 1 | STRING | 0 | “cane” |
| 2 | STRING | 0 | “oca” |
| 3 | LIST | 2 | 0 |
| 4 | LIST | 1 | 2 |

Dunque la lista *animali* si riferisce alla riga 3 dell’instance stack. A sua volta il valore di **inst** di questa lista si riferisce all’indice del suo elemento iniziale presente

nell’*istack*: in questo caso è la stringa “gatto” in posizione 0. Una volta che la lista punta al suo primo elemento, essa viene costruita valutando gli elementi successivi, tramite l’incremento dell’*ip*, fino a quando il loro numero coincide con la lunghezza della lista. Ricapitolando, la lista *animali* è composta dalle liste in posizione 3 e 4 dell’instance stack (infatti la sua **leng** vale 2). La 3 è formata dalle stringhe memorizzate all’indice 0 e 1 dell’*istack*, mentre la 4 dalla sola stringa in posizione 2.

Esempio 2

| variables | TCODE 20 | |
|---------------------------|-------------|--------------|
| x, y, s: int; | 0) VARS 3 | 10) PUSH 2 |
| functions | | 11) JUMP 15 |
| sum(n1:int, n2:int) → int | 1) NEWI | 12) APOP |
| n1 + n2; | 2) NEWI | 13) STOR 2 |
| body | | 4) LOCI 41 |
| x = 41; | 5) STOR 0 | 14) HALT |
| y = 18; | 6) LOCI 18 | 15) FUNC 7 |
| s = sum(x,y); | 7) STOR 1 | 16) LOAD 1 0 |
| | 8) LOAD 0 0 | 17) LOAD 1 1 |
| | 9) LOAD 0 1 | 18) PLUS |
| | | 19) RETN |

Dopo l’istruzione all’indirizzo 9, lo stack delle variabili ha memorizzato x e y coi loro valori 41 e 18, quello dei parametri e delle chiamate di funzioni sono vuoti e l’object stack risulta:

| ostack | | | |
|--------|--------|------|------|
| | DOMAIN | LENG | INST |
| 0 | INT | 0 | 41 |
| 1 | INT | 0 | 18 |

La PUSH agisce sull'*astack* e il *pstack* nella seguente maniera:

| astack | | | |
|--------|-----|----------------------|------|
| | NUM | OBJS | RET |
| 0 | 2 | $2(op) - 2(num) = 0$ | null |

| pstack | | | |
|--------|--------|------|------|
| | DOMAIN | LENG | INST |
| 0 | INT | 0 | 41 |
| 1 | INT | 0 | 18 |

La JUMP passa all'indirizzo 15 e inizializza il campo **ret**:

| astack | | | |
|--------|-----|------|-----|
| | NUM | OBJS | RET |
| 0 | 2 | 0 | 12 |

Quindi si valuta il T-code della funzione *sum()* (indicata con 7 perché l'*oid* da 1 a 6 sono riservati alle built-in) e dopo l'istruzione PLUS si ha la seguente situazione nell'*ostack*:

| ostack | | | |
|--------|--------|------|------|
| | DOMAIN | LENG | INST |
| 0 | INT | 0 | 59 |
| 1 | INT | 0 | 18 |

cioè l'oggetto in 0 (che rappresentava la variabile *x*) è stato sovrascritto col valore della somma di *x* e *y*. Una volta ritornati all'indirizzo 12 e aver eseguito l'APOP, l'*ap* e il *pp*, che ora valevano 0 e 2, tornano ai loro valori precedenti alla chiamata, nello

specifico -1 e 0. Questo affinché gli oggetti presenti attualmente nell’activation stack e nel parameters stack possano venire sovrascritti, dato che hanno solo un’utilità temporanea: si sfrutta in modo efficace la memoria della macchina virtuale evitando che oggetti inutili occupino spazio all’interno degli array. Infine la STOR 2 preleva l’oggetto in cima all’*ostack* e ne assegna il valore alla variabile *s*.

```

1048 int main() {
1049     Tcode *tstat;
1050
1051     start_machine();
1052     printf("\n");
1053     while((tstat = &prog[pc++])->oper != HALT) {
1054         execute(tstat);
1055     }
1056     printf("\n\n");
1057     return 0;
1058 }
```

Figura 6.5: Il codice C del *main()*

Il *main()* dell’interprete invoca la funzione *start_machine()* che si occupa di due aspetti: il primo è l’inizializzazione degli stack (compreso quello della memoria del codice). Il secondo è più impegnativo e consiste nella lettura del file “*IntermediateCode*”, ricevuto in input dal generatore di codice intermedio. Il file viene letto carattere per carattere e il compito della *start_machine()* è quello di trasformare le istruzioni T-code in formato stringa nel formato della struttura analizzata all’inizio di questo capitolo e che rappresenta la memoria del codice (*prog*). Ad esempio, l’array di caratteri “VARS 5” viene così convertito: il valore del campo **Operator** è VARS, derivato tramite la funzione *fromStringToOp()*, che riceve come parametro “VARS”; ad **args[0]** viene assegnato l’intero 5, ricavato con l’uso della *atoi()*, passandole in input “5”. Quindi conclusa la *start_machine()*, l’array *prog* contiene tutte le istruzioni T-code del programma Tofu: a questo punto il *main()* non fa altro che eseguire ogni singola istruzione finché non incontra quella conclusiva (HALT). La funzione

execute() è uno **switch case** che richiama, a seconda dell'operatore, le funzioni auxiliarie opportune per eseguire l'istruzione corrente. Alcune di queste sono già state presentate (vedi figure 6.2, 6.3, 6.4); nella figura sottostante ne vengono mostrate un altro paio.

```

381 void exec_rang() {
382     int start, end, leng;
383     end = pop_obj().inst.ival;
384     start = pop_obj().inst.ival;
385     if(start > end) {
386         leng = 0;
387     }
388     else if (start == end) {
389         leng = 1;
390         push_int(start);
391     }
392     else {
393         leng = end - start + 1;
394         while(start <= end) {
395             push_int(start++);
396         }
397     }
398     push_list(leng);
399 }
515 void exec_getb() {
516     int b;
517     char input[STRING_DIM];
518     scanf("%s", input);
519     if(strcmp("true", input) == 0)
520         b = 1;
521     else if(strcmp("false", input) == 0)
522         b = 0;
523     else
524         error("Attention: wrong boolean format\n");
525     push_bool(b);
526 }
```

Figura 6.6: Il codice C delle funzioni *exec_rang()* ed *exec_getb()*

Come visto per la *exec_last()* e la *exec_init()*, alcune istruzioni del codice intermedio potrebbero dar luogo ad errori run-time. Altri errori run-time si possono verificare con l'inserimento di input dinamico da parte dell'utente, dunque devono essere gestiti durante l'esecuzione della GETI, GETB e GETL.

```

dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./T_code <programs/errori_runtime.tofu
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./tofu

Inserisci una lista di interi: []
Function head applied to empty list
```

Figura 6.7: Errore run-time: funzione *head()* applicata a una lista vuota

```
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./T_code <programs/errori_runtime.tofu
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./tofu

Inserisci un booleano: tr
Attention: wrong boolean format
```

Figura 6.8: Errore run-time: inserito un valore diverso da **true** o **false** per il tipo booleano

Il codice C completo della T-machine (file *interpreter.c*) è reperibile all’Appendice E. L’esecuzione della macchina virtuale produce l’output definito all’interno del programma Tofu: di seguito sono illustrate le esecuzioni dei programmi Tofu *inverti_liste.tofu* e *max_min.tofu*.

```
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ sh makefile.txt
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./T_code <programs/inverti_liste.tofu
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./tofu

Questo programma inverte una lista di liste di stringhe.
Inserisci una lista di stringhe con profondità 2: [[k,j,i,h],[],[g,f,e],[d,c,b,a]]
Lista invertita = [[a,b,c,d],[e,f,g],[],[h,i,j,k]]
```

Figura 6.9: Esecuzione del programma Tofu *inverti_liste.tofu*

```
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./T_code <programs/max_min.tofu
dario01@LAPTOP-RKFLOJ0J:~/Tofu$ ./tofu

Questo programma indica l’elemento minore e quello maggiore di una lista di interi.
Inserisci una lista di interi: [65,43,309,8,112,73,231,999,500,15,784]
Il minimo è: 8
Il massimo è: 999
```

Figura 6.10: Esecuzione del programma Tofu *max_min.tofu*

7 Installazione del software

Il traduttore e l'interprete di Tofu sono pensati per funzionare in ambiente Linux. Tuttavia, anche se l'utente non possiede un computer con sistema operativo (SO) Linux, non c'è nessun problema: accanto al proprio SO nativo (ad esempio Windows o macOS) è possibile installare Ubuntu su una partizione del disco. In questo modo, durante la fase di avvio del PC (bootstrap), verrà chiesto di scegliere quale SO si desidera utilizzare (dual boot). Per l'installazione di Ubuntu si possono seguire le istruzioni alla pagina web <https://wiki.ubuntu-it.org/Installazione/InstallareUbuntu>, altrimenti si può consultare altro materiale presente in rete. Se si ha Windows, un'alternativa a questa modalità è l'installazione dell'ambiente terminale Ubuntu sulla propria macchina, senza la necessità del dual boot: ciò è reso possibile dal WSL (Windows Subsystem for Linux). Al sito <https://ubuntu.com/tutorials/install-ubuntu-on-wsl2-on-windows-10#1-overview> spiegano cosa fare, ma, come prima, esistono anche altre pagine web che illustrano i passaggi richiesti.

Il software presentato dal capitolo 4 al capitolo 6 è composto dai seguenti nove file:

| | | | |
|---|------------------|--------------------|-------|
|  def.h | 22/01/2023 17:44 | C Header File | 2 KB |
|  gen.c | 29/11/2022 13:16 | File di origine C | 19 KB |
|  interpreter.c | 29/11/2022 13:16 | File di origine C | 20 KB |
|  lexer.lex | 15/11/2022 21:17 | File LEX | 2 KB |
|  makefile.txt | 13/09/2022 18:43 | Documento di testo | 1 KB |
|  parser.y | 16/11/2022 15:44 | File Y | 6 KB |
|  readme.txt | 08/11/2022 16:38 | Documento di testo | 2 KB |
|  semantics.c | 11/01/2023 23:38 | File di origine C | 26 KB |
|  tree.c | 22/11/2022 13:58 | File di origine C | 2 KB |

Gli unici rimasti da analizzare sono il *makefile.txt* e il *readme.txt*, che costituisce una guida per l'utente nell'uso del software.

```
1 - Per avere a disposizione il traduttore e l'interprete di Tofu bisogna compilare
2   il file makefile.txt da linea di comando nel seguente modo:
3
4   sh makefile.txt
5
6
7 - Ogni volta che si scrive un nuovo programma Tofu o se ne modifica uno
8   esistente, bisogna semplicemente eseguire i comandi:
9
10  ./T_code <program.tofu
11  ./tofu
12
13
14 - Se invece il programma non viene modificato ed è sempre lo stesso,
15   allora basta eseguire il comando:
16
17  ./tofu
18
19
20 - Comando per convertire il carattere eof (end of file) da Windows a Unix:
21
22  tr -d '\15\32' < winfile.txt > unixfile.txt
23
24 -----
25
26 File richiesti per il funzionamento del software:
27
28 -def.h
29 -tree.c
30 -lexer.lex      (analizzatore lessicale)
31 -parser.y       (analizzatore sintattico)
32 -semantics.c   (analizzatore semantico)
33 -gen.c          (generatore di codice intermedio)
34 -interpreter.c (interprete/macchina virtuale)
35
36 -----
37
38 LEGENDA dei comandi da terminale richiesti dal software:
39
40   output:  input
41
42   comandi linux
43
44
45
46 1) lexer.c : lexer.lex
47
48  | flex -o lexer.c lexer.lex
49
50
51 2) parser.h, parser.c, parser.output, parser.dot : parser.y
52
53  | bison -dvg -o parser.c parser.y
54
55
56 3) lexer.o : lexer.c, def.h, parser.h
57
58  | cc -g -c lexer.c
59
60
```

```

61 4) parser.o, parser.pdf : parser.c, def.h, parser.dot
62
63     cc -g -c parser.c
64     dot -Tpdf -o parser.pdf parser.dot
65
66
67 5) tree.o : tree.c, def.h
68
69     cc -g -c tree.c
70
71
72 6) gen.o : semantics.c gen.c, def.h, parser.h
73
74     cc -g -c gen.c
75
76
77 7) T_code : lexer.o, parser.o, tree.o, gen.o
78
79     cc -g -o T_code lexer.o parser.o tree.o gen.o
80
81
82 8) interpreter.o : interpreter.c, def.h
83
84     cc -g -c interpreter.c
85
86
87 9) tofu : interpreter.o
88
89     cc -g -o tofu interpreter.o
90
91
92 10) AbstractTree.txt, SymbolTable.txt, IntermediateCode.txt : T_code, program.tofu
93
94     ./T_code <program.tofu
95
96
97 11) output del programma Tofu : tofu
98
99     ./tofu

```

Figura 7.1: Contenuto del file *readme.txt*

Il comando **sh** in Linux (a riga 4 in figura 7.1) permette di eseguire una sequenza di comandi presenti in un file di testo, anziché doverli eseguire uno ad uno. Il file *makefile.txt* contiene in forma più compatta i comandi Linux indicati dal numero 1) al numero 9) all'interno del *readme.txt*. Eseguendo l'unico comando **sh makefile.txt** l'utente ha immediatamente a disposizione i file eseguibili *T_code* (output del generatore del codice intermedio) e *tofu* (output dell'interprete). Il comando a riga 22 della figura 7.1, serve solo se si sta usando WSL e si creano dei file di testo in Windows che poi devono essere convertiti in file di testo Unix, visto che il carattere

di **eol** (end of file) è differente. Per eseguire un programma Tofu, denominato ad esempio *prova.tofu*, prima si genera il suo T-code tramite **./T_code <prova.tofu** (riga 10) e poi si interpreta il suo T-code mediante il comando **./tofu** (riga 11): ora l'utente sarà in grado di visualizzare l'output di *prova.tofu* e, se richiesti, inserire input da tastiera.

Infine il *readme.txt* specifica gli output e gli input prodotti dai comandi dall' 1) all' 11) (righe 46-99).

- L'opzione **-c** indica che il file sorgente C viene compilato senza collegamento, producendo un file oggetto (con estensione .o).
- Il flag **-g** genera informazioni di debug che vengono usate dal GDB debugger.
- **-o** denota il nome del file di output.
- Nel comando **bison -dvg -o parser.c parser.y** (riga 53)
 - il flag **-d** (header) produce il file *parser.h*;
 - **-v** (verbose) genera la descrizione testuale della tabella di parsing LALR(1) (file *parser.output*);
 - **-g** (graphic) ha come output la rappresentazione dell'automa di parsing LALR(1) nel linguaggio *dot* (file *parser.dot*).
- Il comando **dot -Tpdf -o parser.pdf parser.dot** (riga 64) converte il file *parser.dot* nel pdf *parser.pdf*.

8 Conclusioni

Si è cominciato definendo che cosa sono e perché esistono i linguaggi di programmazione, vedendo sinteticamente la loro classificazione, la loro specifica e le loro possibili implementazioni. Si è visto che la tipologia scelta per un linguaggio risulta cruciale in fase di progettazione, come lo è in altrettanto modo una sua analisi a livello lessicale, sintattico e semantico. Questi concetti sono alla base dell’ideazione e dello sviluppo del nuovo linguaggio di programmazione **Tofu**. Infatti il cuore della relazione consiste nell’illustrare nei minimi dettagli tutti i passaggi richiesti per la sua realizzazione: dalla sua progettazione, all’implementazione del traduttore; dal codice intermedio così generato, alla sua interpretazione da parte della macchina virtuale, che porta in ultima istanza all’esecuzione del programma scritto in **Tofu**. Il linguaggio presentato, come detto in precedenza, è di tipo *pseudocompilato*, tuttavia si potrebbe modificarlo in uno di tipo *interpretato*. A tale scopo, il generatore di codice intermedio (file *gen.c*) non servirebbe più e bisognerebbe apportare i giusti cambiamenti all’interprete (file *interpreter.c*), affinché attinga le informazioni necessarie direttamente dall’albero sintattico e dall’analizzatore semantico. Si evidenzia che, se avvenisse questa modifica, la specifica di **Tofu** risulterebbe immutata. Eventuali sviluppi futuri potrebbero proprio riguardare l’ampliamento di quest’ultima, introducendo ad esempio altri tipi di dati (**float**), ulteriori operazioni (la potenza, il modulo, ecc...), altre funzioni built-in (invertire una lista, cancellare o trovare un elemento al suo interno, ecc...) e/o ulteriori costrutti (**for**, **while**, ecc...). In questa maniera **Tofu**, che al momento è praticamente un linguaggio “giocattolo” (come dice il nome stesso), in quanto dotato di una specifica scheletrica, potrebbe diventare più “potente” e interessante: dunque adatto per lo sviluppo di applicazioni software.

Bibliografia e Sitografia

- [1] https://it.wikipedia.org/wiki/Linguaggio_di_programmazione.
- [2] <https://it.wikipedia.org/wiki/Software>.
- [3] https://it.wikipedia.org/wiki/Programmazione_imperativa
- [4] https://it.wikipedia.org/wiki/Programmazione_orientata_agli_oggetti
- [5] https://it.wikipedia.org/wiki/Programmazione_funzionale
- [6] https://it.wikipedia.org/wiki/Programmazione_logica
- [7] https://www.edatlas.it/scarica/informatica/info_java/Capitolo4/2LinguaggiPuriIbridi.pdf
- [8] Gian Franco Lamperti, slides del corso “*Linguaggi di Programmazione*”, reperibili al sito <https://gianfranco-lamperti.unibs.it/lp/lp.html>, Università degli Studi di Brescia
- [9] <https://www.phpcodewizard.it/antoniolamorgese/come-creare-un-linguaggio-di-programmazione/>
- [10] <https://texandman.forumfree.it/?t=44287546>
- [11] <https://vitolavecchia.altervista.org/differenza-tra-linker-e-loader-informatica/>
- [12] <https://vitolavecchia.altervista.org/differenza-tra-tempo-di-compilazione-e-di-esecuzione/>
- [13] <https://it.wikipedia.org/wiki/Compilatore>
- [14] https://it.wikipedia.org/wiki/Linguaggio_compilato

- [15] [https://it.wikipedia.org/wiki/Interprete_\(informatica\)](https://it.wikipedia.org/wiki/Interprete_(informatica))
- [16] <https://www.freecodecamp.org/italian/news/linguaggi-di-programmazione-interpretati-e-compilati-qual-e-la-differenza>
- [17] Gian Franco Lamperti, slides del corso “*Tecnologie dei linguaggi artificiali*”, reperibili al sito <https://gianfranco-lamperti.unibs.it/tl/tl.html>, Università degli Studi di Brescia
- [18] <http://lacam.di.uniba.it/~nico/corsi/lingpro/materiale/LP-AnalisiSintattica.pdf>

Appendice A

A.1 Espressioni Regolari

Notazione:

- Due caratteri consecutivi senza spazio indicano concatenazione
- $|$ = indica alternative che si escludono a vicenda: $a | b$
- ϵ = indica una stringa lessicale vuota
- $*$ = ripetizione zero o più volte di una stringa lessicale: $(abc)^*$

Notazione delle espressioni regolari estese:

- $^+$ = ripetizione una o più volte di una stringa lessicale: $(ab | cd)^+$
- $.$ = indica qualsiasi carattere dell'alfabeto Σ : $.bde$.
- $[...]$ = indica un range di caratteri: $[0-9]$
- \sim = indica qualsiasi carattere al di fuori di un certo insieme: $\sim a$ (equivale a $\Sigma - \{a\}$)
- ?: indica l'opzionalità del carattere: $(+ | -)? [0-9]$

Esempio:

Dato un alfabeto $\Sigma = \mathbf{a}, \mathbf{b}, \mathbf{c}$, si chiede di specificare l'espressione regolare delle stringhe che contengono esattamente tre **b** separati fra loro da altri caratteri.

$$(\mathbf{a} | \mathbf{c})^* \mathbf{b} (\mathbf{a} | \mathbf{c})^+ \mathbf{b} (\mathbf{a} | \mathbf{c})^+ \mathbf{b} (\mathbf{a} | \mathbf{c})^*$$

Notazione delle definizioni regolari:

$\text{simbolo}_1 \rightarrow \text{expr_reg}_1$

...

$\text{simbolo}_n \rightarrow \text{expr_reg}_n$

Esempio:

Specificare la definizione regolare relativa ai simboli lessicali **realconst** (costante reale) e **id** (identificatore), dove:

- Una costante reale ha segno opzionale, una parte intera obbligatoria e una decimale opzionale;
- Un identificatore inizia con un carattere alfabetico ed è seguito da una sequenza (anche vuota) di caratteri alfanumerici e ha massimo quattro caratteri.

```
lettera → [A-Za-z]
cifra → [0-9]
alfanum → lettera | cifra
cifra-iniziale → [1-9]
segno → '+'
intero → cifra-iniziale cifra* | 0
decimale → cifra+
realconst → segno? intero ('.' decimale)?
id → lettera alfanum? alfanum? alfanum?
```

A.2 Lex

Nel paragrafo 3.2.1 è stato detto che cosa è Lex e come è strutturato a grandi linee.

Più nello specifico, gli identificatori built-in (interni) di Lex sono:

- **yylex()**: la funzione che svolge l'analisi lessicale descritta dal programma;
- **yytext**: la stringa lessicale (lexeme) che sta venendo analizzata;
- **yylen**: la lunghezza, in termini di caratteri, di yytext (`strlen(yytext)`);

- **yyin**: il file dato in input al programma (di default è stdin);
- **yyout()**: il file prodotto in output dal programma (di default è stdout);
- **input()**: un singolo carattere in input;
- **ECHO**: svolge l'azione di default di stampa di yytext su yyout.

Di seguito sono chiariti gli aspetti riguardanti la notazione di Lex, con enfasi sulle regole di traduzione:

- Le definizioni regolari si scrivono come descritto sopra (sezione A.1), con l'unica differenza che anziché usare la →, si lascia uno o più spazi vuoti e che il metacarattere ~ è sostituito dal metacarattere ^ ;
- **%option noyywrap**: è l'opzione di terminazione, cioè indica che, una volta finito di leggere il programma, l'analisi deve terminare, senza iniziare nuovamente una rilettura del programma;
- le librerie, le variabili, le costanti e le funzioni vengono specificate come nel linguaggio C;
- **simbolo {regola}**: è la notazione delle regole di traduzione.
 - Se il **simbolo** è tra le dichiarazioni regolari, lo si include nelle graffe {}, se è una stringa lessicale la si include fra gli apici "", se è una keyword (parola chiave) del linguaggio non la si include in niente.
 - Se la **regola** è vuota, cioè indicata dal solo carattere punto e virgola ; , Lex ignorerà il simbolo collegato durante la sua analisi (di solito usata per la spaziatura e i commenti).
 - La regola può includere codice C;

- Infine, all'interno di una regola, non vuota, viene specificato il valore di ritorno associato al simbolo, che permetterà di identificarlo in maniera univoca durante le successive fasi di analisi: infatti tale valore costituisce un input del parser.

Esempio di programma Lex per la specifica della BNF mostrata di seguito, dove un identificatore **id** è una sequenza di lettere minuscole, eventualmente separate da un solo underscore, e la costante booleana **boolconst** ha valori **true** o **false**.

```
%{
#include <stdlib.h>
#include "def.h" /* encoding of lexical symbols */
Lexval lexval;
%}
delimiter [ \t\n]
spacing {delimiter}+
lowercase [a-z]
id {lowercase}+('_'{lowercase})*
boolconst false | true
sugar [::]
%%
{spacing} ;
{sugar} {return(yytext[0]);}
display {return(DISPLAY);}
int {return(INT);}
string {return(STRING);}
if {return(IF);}
then {return(THEN);}
else {return(ELSE);}
{boolconst} {lexval.ival = (yytext[0]=='f' ? 0 : 1); return(BOOLCONST);}
{id} {lexval.sval = newstring(yytext); return(ID);}
.
{return(ERROR);}

char *newstring(char *s)
{
    char *p = malloc(strlen(s)+1);
    strcpy(p,s);
    return(p);
}
```

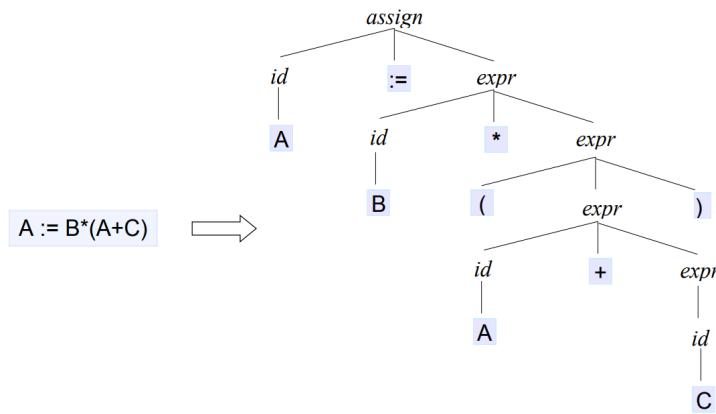
$$\begin{aligned} \text{program} &\rightarrow \text{stat-list} \\ \text{stat-list} &\rightarrow \text{stat} ; \text{stat-list} \mid \epsilon \\ \text{stat} &\rightarrow \text{def-stat} \mid \text{if-stat} \mid \text{display} \\ \text{def-stat} &\rightarrow \text{id} : \text{type} \\ \text{type} &\rightarrow \text{int} \mid \text{string} \\ \text{if-stat} &\rightarrow \text{if} \text{ expr then stat else stat} \\ \text{expr} &\rightarrow \text{boolconst} \end{aligned}$$

Altri esempi sono la figura 3.7 e il *lexer.lex* di Tofu (figura 5.2).

Appendice B

B.1 BNF ed EBNF

Illustrazione dell'albero sintattico della frase mostrata in figura 2.5. Leggendo le foglie dell'albero da sinistra verso destra si è in grado di ricostruire la frase di partenza.



La BNF è composta da una serie di produzioni. Una produzione specifica il significato sintattico di un'astrazione (simbolo nonterminale) tramite l'uso di altri simboli nonterminali o stringhe lessicali (simboli terminali).

Esempio di una produzione relativa all'operazione di assegnamento:

$$assign \rightarrow var := expr$$

Parafrasi della regola (produzione):

Un'istanza dell'astrazione *assign* è definita come un'istanza dell'astrazione *var*, seguita dalla stringa lessicale ' $:=$ ', seguita da un'istanza dell'astrazione *expr*.

Notazione della BNF:

- La concatenazione è indicata da due o più simboli terminali/nonterminali consecutivi
- $|$ = indica alternative che si escludono a vicenda
- ϵ = indica una stringa lessicale vuota

Notazione della EBNF:

- $\{...\}$ = ripetizione di zero o più volte di un nonterminale/terminale
- $\{...\}^+$ = ripetizione una o più volte di un nonterminale/terminale
- $[...]$ = indica l'opzionalità del nonterminale/terminale
- $(...)$ = unito al metacarattere $|$, indica disgiunzione di un nonterminale/terminale

Esempio:

Dato il seguente linguaggio per tavole

| R | | | S | |
|---|------|-------|---|----------------------|
| A | B | C | D | E |
| 3 | alfa | true | def R (A: integer, B: string, C: boolean) def S (D: integer, E: string) R := {(3, "alfa", true)(5, "beta", false)} S := {(125, "sole")(236, "luna")} | 125 sole 236 luna |
| 5 | beta | false | | |

specificare la BNF:

```

program → stat-list
stat-list → stat-list stat | stat
stat → def-stat | assign-stat
def-stat → def id ( def-list )
def-list → def-list, domain-decl | domain-decl
domain-decl → id : domain
domain → integer | string | boolean
assign-stat → id := { tuple-list }
tuple-list → tuple-list tuple-const | ε
tuple-const → ( simple-const-list )
simple-const-list → simple-const-list, simple-const | simple-const
simple-const → intconst | strconst | boolconst

```

Definiamo ora la sintassi del medesimo linguaggio tramite EBNF:

```

program → {stat }+
stat → def-stat | assign-stat
def-stat → def id "(" def-list ")"
def-list → domain-decl {, domain-decl }
domain-decl → id : domain
domain → integer | string | boolean
assign-stat → id := "{" {tuple-const} "}"
tuple-const → "(" simple-const {, simple-const} ")"
simple-const → intconst | strconst | boolconst

```

B.2 BNF del linguaggio Tofu

```

program → var-section func-section body-section
var-section → variables var-decl-list | ε
var-decl-list → var-decl var-decl-list | var-decl
var-decl → id-list : type ;
id-list → id , id-list | id
type → int | string | bool | [ type ]

```

$\text{func-section} \rightarrow \mathbf{functions} \text{ func-decl-list} \mid \epsilon$
 $\text{func-decl-list} \rightarrow \text{func-decl} \text{ func-decl-list} \mid \text{func-decl}$
 $\text{func-decl} \rightarrow \mathbf{id} (\text{opt-formal-list}) \rightarrow \text{type} \text{ expr-decl}$
 $\text{opt-formal-list} \rightarrow \text{formal-list} \mid \epsilon$
 $\text{formal-list} \rightarrow \text{formal-decl}, \text{ formal-list} \mid \text{formal-decl}$
 $\text{formal-decl} \rightarrow \mathbf{id} : \text{type}$
 $\text{expr-decl} \rightarrow \text{guard-expr-list} \mid \text{expr} ;$
 $\text{guard-expr-list} \rightarrow \text{guard-expr} \text{ guard-expr-list} \mid \text{guard-expr} \text{ guard-last}$
 $\text{guard-expr} \rightarrow \mathbf{I} \text{ expr} \rightarrow \text{expr} ;$
 $\text{guard-last} \rightarrow \mathbf{I} \text{ otherwise} \rightarrow \text{expr} ;$
 $\text{expr} \rightarrow \text{expr} \text{ bool-op} \text{ bool-term} \mid \text{bool-term}$
 $\text{bool-op} \rightarrow \mathbf{and} \mid \mathbf{or}$
 $\text{bool-term} \rightarrow \text{comp-term} \text{ comp-op} \text{ comp-term} \mid \text{comp-term}$
 $\text{comp-op} \rightarrow == \mid != \mid > \mid >= \mid < \mid <=$
 $\text{comp-term} \rightarrow \text{comp-term} \text{ add-op} \text{ term} \mid \text{term}$
 $\text{add-op} \rightarrow + \mid - \mid ++$
 $\text{term} \rightarrow \text{term} \text{ mul-op} \text{ factor} \mid \text{factor}$
 $\text{mul-op} \rightarrow * \mid /$
 $\text{factor} \rightarrow \text{unary-op} \text{ factor} \mid (\text{expr}) \mid \mathbf{id} \mid \mathbf{intconst} \mid \mathbf{strconst} \mid \mathbf{boolconst} \mid [\text{opt-expr-list}] \mid \text{if-expr} \mid \text{func-call} \mid \text{input} \mid \text{output} \mid \text{range}$
 $\text{unary-op} \rightarrow - \mid \mathbf{not}$
 $\text{opt-expr-list} \rightarrow \text{expr-list} \mid \epsilon$
 $\text{expr-list} \rightarrow \text{expr}, \text{expr-list} \mid \text{expr}$
 $\text{if-expr} \rightarrow \mathbf{if} \text{ expr} \mathbf{then} \text{ expr} \mathbf{else} \text{ expr} \mathbf{end}$
 $\text{input} \rightarrow ? (\text{type})$
 $\text{output} \rightarrow ! (\text{expr})$
 $\text{func-call} \rightarrow \mathbf{id} (\text{opt-expr-list})$
 $\text{range} \rightarrow [\text{expr} . . \text{expr}]$

 $\text{body-section} \rightarrow \mathbf{body} \text{ stat-list}$
 $\text{stat-list} \rightarrow \text{stat} ; \text{stat-list} \mid \text{stat} ;$
 $\text{stat} \rightarrow \text{assign-stat} \mid \text{show-stat}$
 $\text{assign-stat} \rightarrow \mathbf{id} = \text{expr-decl}$
 $\text{show-stat} \rightarrow \mathbf{show} \text{ expr-decl}$

B.3 EBNF del linguaggio Tofu

program \rightarrow *var-section* *func-section* *stat-list*
var-section \rightarrow { *var-decl* }
var-decl \rightarrow *id-list type*
id-list \rightarrow { **id** }⁺
type \rightarrow **atomic-type** | *list-type* in cui **atomic-type** qualificato come: INT | STRING | BOOL
list-type \rightarrow *type*
func-section \rightarrow { *func-decl* }
func-decl \rightarrow **id** *opt-formal-list type EXPR*
opt-formal-list \rightarrow { *formal-decl* }
formal-decl \rightarrow **id** *type*
stat-list \rightarrow { *assign-stat* | *show-stat* }⁺
assign-stat \rightarrow **id** **EXPR**
show-stat \rightarrow **EXPR**

in cui **EXPR** = (*logic-expr* | *comp-expr* | *math-expr* | *neg-expr* | **id** | **intconst** | **strconst** | **boolconst** | *opt-expr-list* | *if-expr* | *func-call* | *input* | *output* | range | *guard-expr* | *guard-last*)

in cui la qualifica del nonterminale è stabilita nel seguente modo:

logic-expr : AND | OR
comp-expr : EQ | NE | '>' | GE | '<' | LE
math-expr : '+' | '-' | '*' | '/' | APP
neg-expr : '-' | NOT

logic-expr \rightarrow **EXPR** **EXPR**
comp-expr \rightarrow **EXPR** **EXPR**
math-expr \rightarrow **EXPR** **EXPR**
neg-expr \rightarrow **EXPR**
opt-expr-list \rightarrow { **EXPR** }
if-expr \rightarrow **EXPR** **EXPR** **EXPR**
func-call \rightarrow **id** { **EXPR** }
input \rightarrow *type*
output \rightarrow **EXPR**
range \rightarrow **EXPR** **EXPR**
guard-expr \rightarrow **EXPR** **EXPR**
guard-last \rightarrow **EXPR**

B.4 Yacc

Come accennato nel paragrafo 3.2.2, la struttura di Yacc è identica a quella di Lex.

Nelle dichiarazioni blackbox però, oltre a includere file e definire costanti e variabili (anche esterne), si possono dichiarare delle variabili il cui tipo è stato definito in un file separato:

```
#define YYSTYPE TYPE
```

Le dichiarazioni whitebox specificano i simboli terminali della grammatica (gli stessi ritornati dalle regole di traduzione di Lex) e sono introdotte da **%token**.

Le regole di traduzione specificano le regole di produzione e le loro azioni semantiche associate e si presentano nel seguente modo:

$$\begin{aligned} A : & \alpha_1 \{ \text{azione 1} \} \\ | & \alpha_2 \{ \text{azione 2} \} \\ & \dots \\ | & \alpha_n \{ \text{azione n} \} \\ ; & \end{aligned}$$

- **A** è il simbolo nonterminale a cui viene associata la regola sintattica specificata nella parte destra (dopo il metacarattere `:`). Il primo nonterminale di Yacc è l'assioma.
- Un nonterminale deve avere una regola associata; tuttavia può averne anche più di una: le alternative sono separate dal metacarattere `|`. La terminazione delle regole associate a un nonterminale è rappresentata dal metacarattere `;`.
- Le regole di produzione ($\alpha_1, \dots, \alpha_n$) sono esattamente quelle espresse dalla BNF del linguaggio; i simboli terminali della BNF sono indicati tra gli apici singoli “ o coi nomi dei token.

- In Yacc vengono usate delle pseudo-variabili per referenziare i valori degli attributi semanticici: **\$\$** per l'attributo (il nonterminale) di sinistra e **\$i** per l'i-esimo attributo (nonterminale/terminale) di destra.
- Per una regola di produzione possono essere specificate, all'interno delle graffe, più azioni semantiche separate dal metacarattere ; . Se una regola di produzione non dichiara nessuna azione, significa che verrà eseguita quella di default (**\$\$ = \$1**). Le azioni semantiche sono frammenti di codice C, che possono richiamare le funzioni ausiliarie dichiarate nell'ultima sezione di Yacc e che, di solito, vengono usate per costruire l'albero sintattico del linguaggio. Infatti, partendo dalla radice, i nonterminali vengono collegati gerarchicamente fra loro fino ad arrivare a dei terminali, che rappresentano le foglie dell'albero. Questo viene fatto tramite le variabili di tipo **\$** e l'uso dei puntatori.

Esempi:

- **\$\$ = \$1** significa che al nonterminale di sinistra viene passato il valore del primo simbolo di destra: cioè solo il simbolo di destra viene incluso nell'albero sintattico, mentre quello di sinistra ne è escluso.
- **\$\$ → p1 = \$2** (assumendo che **p1** sia un puntatore) indica che il secondo simbolo di destra rappresenta il figlio del nonterminale di sinistra all'interno dell'albero sintattico.
- In Yacc si possono specificare *azioni embedded*: azioni semantiche dichiarate all'interno di una produzione ogni volta che è necessario eseguire del codice prima del riconoscimento completo della stessa. Tecnica tipicamente usata quando si ha a che fare con gli identificatori di un linguaggio.
- Se viene specificata una grammatica ambigua, i conflitti vengono individuati e risolti nel miglior modo possibile da Yacc: tra un'operazione di spostamento

e una di riduzione in conflitto, viene scelta quella di spostamento (spesso va bene); tra due riduzioni, viene eseguita la prima delle due che è stata dichiarata in Yacc (tuttavia in questa casistica bisogna ridefinire la grammatica).

- **yyval** è la variabile contenente il valore lessicale dei token (assegnato da Lex).

Infine, nella sezione delle funzioni ausiliarie vengono dichiarate funzioni C necessarie per completare l'analisi sintattica. In particolare, il parsing del linguaggio viene svolto richiamando la funzione **yyparse()** che restituisce **0** se il programma è sintatticamente corretto, altrimenti **1**. La **yyparse()** richiama la **yylex()** durante la sua esecuzione; inoltre, quando ritorna 1, significa che c'è un errore nel programma e per segnalarlo si serve della **yyerror()**.

Esempi di programmi Yacc sono la figura 3.12 e il *parser.y* di Tofu (figura 5.4).

Appendice C

Il codice C completo dell'analizzatore semantico di Tofu (file *semantics.c*).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "def.h"
5 #include "parser.h"
6
7 #define TOT 1031
8 #define TOT_FUNC 191
9 #define OFFSET 20
10 #define SHIFT 4
11 #define UNO 1
12 #define MENO_UNO -1
13 #define TRUE 1
14 #define FALSE 0
15 #define NOTHING ""
16 #define NUM_BUILT_IN 6
17 #define EMPTY "empty"
18 #define LENGTH "length"
19 #define HEAD "head"
20 #define TAIL "tail"
21 #define LAST "last"
22 #define INIT "init"
23
24 extern Pnode root;
25
26 typedef enum {
27     D_INT,
28     D_STRING,
29     D_BOOL,
30     D_LIST
31 } Domain;
32
33 typedef enum {
34     A_UNKNOWN,
35     A_INT,
36     A_STRING,
37     A_BOOL
38 } Atomic;
39
40 typedef struct {
41     Domain domain;
42     int depth;
43     Atomic atomic;
44 } Type;
45
46 typedef enum {
47     VAR,
48     FUNC,
49     PARAM
50 } Class;
51
52 typedef struct {
53     int num;
54     struct stable *params;
55 } Formals;
56
57 typedef struct stable {
```

```

58     char *name;
59     Class classe;
60     int oid;
61     Type tipo;
62     struct stable *local;
63     Formals formals;
64     struct stable *next;
65
66 } SymbolTable;
67
68 SymbolTable symbolTable[TOT];
69 int varOid, paramOid = 0;
70 int funcOid = 1;
71 int funcIndex[TOT_FUNC];
72 int checkVar = FALSE;
73 int nextFunc = FALSE;
74 char *msg_error = "Si Inizializza la stringa msg_error";
75
76 int hash (char* id, int dim) {
77     int h=0;
78     for(int i=0; id[i] != '\0'; i++)
79         h = ((h << SHIFT) + id[i]) % dim;
80     return h;
81 }
82
83 int binarySearch(int n, int num) {
84     int i, start = 0, end = n-1;
85     do {
86         i = (start + end)/2;
87         if(i==num)
88             return i;
89         else if(i < num)
90             start = i+1;
91         else
92             end = i-1;
93     } while(start <= end);
94     return -1;
95 }
96
97 int maxArray(int array[]) {
98     int max = array[0];
99     for(int i=1; i < TOT_FUNC && array[i] != MENU_UNO; i++) {
100         if(array[i] > max) {
101             max = array[i];
102         }
103     }
104     return max;
105 }
106
107 void error(char *msg) {
108     printf("%s", msg);
109     exit(-1);
110 }
111
112 Domain defineDomain(Pnode p) {
113     if(p->type == T_INT || p->type == T_INTCONST)
114         return D_INT;
115     else if(p->type == T_STRING || p->type == T_STRCONST)
116         return D_STRING;
117     else if (p->type == T_BOOL || p->type == T_BOOLCONST)
118         return D_BOOL;
119     else if(p->value.ival == NLIST_TYPE)
120         return D_LIST;
121 }
122
123 Atomic defineAtomic(Pnode p) {
124     if(p->type == T_INT || p->type == T_INTCONST)
125         return A_INT;

```

```

126     |     else if(p->type == T_STRING || p->type == T_STRCONST)
127     |     return A_STRING;
128     |     else if (p->type == T_BOOL || p->type == T_BOOLCONST)
129     |     return A_BOOL;
130     |     else
131     |     return A_UNKNOWN;
132 }
133
134 Atomic fromDomainToAtomic(Domain d) {
135     |     if(d == D_INT)
136     |     return A_INT;
137     |     else if(d == D_STRING)
138     |     return A_STRING;
139     |     else if (d == D_BOOL)
140     |     return A_BOOL;
141 }
142
143 Domain fromAtomicToDomain(Atomic a) {
144     |     if(a == A_INT)
145     |     return D_INT;
146     |     else if(a == A_STRING)
147     |     return D_STRING;
148     |     else if (a == A_BOOL)
149     |     return D_BOOL;
150 }
151
152 Type defineType(Pnode p) {
153     Type type;
154     type.domain = defineDomain(p);
155     type.depth = 0;
156     if(type.domain == D_LIST) {
157         while (p->value.ival == NLIST_TYPE) {
158             type.depth++;
159             p = p->p1->p1;
160         }
161         type.atomic = defineAtomic(p);
162     } else {
163         type.atomic = A_UNKNOWN;
164     }
165     return type;
166 }
167
168 void insert(SymbolTable *sTable, char *name, Class classe, int oid, Type tipo,
169     |     struct stable *local, int num, struct stable *params, struct stable *next) {
170     sTable->name = name;
171     sTable->classe = classe;
172     sTable->oid = oid;
173     sTable->tipo = tipo;
174     sTable->local = local;
175     sTable->formals.num = num;
176     sTable->formals.params = params;
177     sTable->next = next;
178 }
179
180 SymbolTable *buildLocal(int length) {
181     struct {
182         |     SymbolTable *st;
183     } Local;
184
185     Local.st = malloc(length * sizeof (SymbolTable));
186     for(int i=0; i < length; i++) {
187         Local.st[i].name = NULL;
188     }
189     return Local.st;
190 }
191
192 int numParam(Pnode param) {
193     int num = 0;
194     while(param != NULL) {
195         num++;

```

```

196     |     param = param->p3;
197   }
198   return num;
199 }
200
201 void whereInsert(SymbolTable sTable[], int index, char *name, Class c, Type t) {
202   if(sTable[index].name == NULL) {
203     if(c == VAR) {
204       insert(&sTable[index], name, c, varOid, t, NULL, MENO_UNO, NULL, buildLocal(UNO));
205       varOid++;
206     } else {
207       insert(&sTable[index], name, c, paramOid, t, NULL, MENO_UNO, NULL, buildLocal(UNO));
208     }
209     sTable[index].next->name = NOTHING;
210   } else {
211     if(c == VAR) {
212       insert(sTable[index].next, name, c, varOid, t, NULL, MENO_UNO, NULL, buildLocal(UNO));
213       varOid++;
214     } else {
215       insert(sTable[index].next, name, c, paramOid, t, NULL, MENO_UNO, NULL, buildLocal(UNO));
216     }
217     sTable[index].next->next->name = NOTHING;
218   }
219 }
220
221 void whereInsertFunc(SymbolTable sTable[], int index, char *name, Class c, Type t, int numParam) {
222   if(sTable[index].name == NULL) {
223     insert(&sTable[index], name, c, funcOid, t, buildLocal(TOT_FUNC), numParam, buildLocal(numParam), buildLocal(UNO));
224     sTable[index].next->name = NOTHING;
225     funcOid++;
226   } else {
227     insert(sTable[index].next, name, c, funcOid, t, buildLocal(TOT_FUNC), numParam, buildLocal(numParam), buildLocal(UNO));
228     sTable[index].next->next->name = NOTHING;
229     funcOid++;
230   }
231 }
232
233 char* describeClass(Class c) {
234   char *nome_classe;
235   if(c == VAR)
236     nome_classe = "VARIABLE";
237   else if(c == FUNC)
238     nome_classe = "FUNCTION";
239   else
240     nome_classe = "PARAMETER";
241
242   return nome_classe;
243 }
244
245 char* describeDomain(Domain d) {
246   char *nome_dominio;
247   if(d == D_INT)
248     nome_dominio = "INT";
249   else if(d == D_STRING)
250     nome_dominio = "STRING";
251   else if(d == D_BOOL)
252     nome_dominio = "BOOL";
253   else
254     nome_dominio = "LIST";
255
256   return nome_dominio;
257 }
258
259 char* describeAtomic(Atomic a) {
260   char *nome_atomico;
261   if(a == A_INT)
262     nome_atomico = "INT";
263   else if(a == A_STRING)
264     nome_atomico = "STRING";

```

```

265     else if(a == A_BOOL)
266         nome_atomico = "BOOL";
267     else
268         nome_atomico = "UNKNOWN";
269
270     return nome_atomico;
271 }
272
273 int lookup(char *id, SymbolTable st[], int dim) {
274     int ok = FALSE;
275     int h = binarySearch(dim, hash(id, dim));
276     if(st[h].name != NULL) {
277         if(strcmp(st[h].name, id) == 0 || strcmp(st[h].next->name, id) == 0) {
278             ok = TRUE;
279         }
280     }
281     return ok;
282 }
283
284 void domainEqual(Domain d1, Domain d2) {
285     if(d1 != d2) {
286         printf("%s", msg_error);
287         fprintf(stderr, "Error: incompatible types; must be all %s or %s \n",
288             describeDomain(d1), describeDomain(d2));
289         exit(-1);
290     }
291
292     void typeEqual(Type t1, Type t2) {
293         if(t1.domain != t2.domain) {
294             printf("%s", msg_error);
295             fprintf(stderr, "Error: incompatible types; must be all %s or %s \n",
296                 describeDomain(t1.domain), describeDomain(t2.domain));
297             exit(-1);
298         }
299         else if(t1.domain == D_LIST) {
300             if(t1.atomic != A_UNKNOWN && t2.atomic != A_UNKNOWN && t1.depth != t2.depth) {
301                 printf("%s", msg_error);
302                 fprintf(stderr, "Error: incompatible list types; must both have depth %d or %d \n",
303                     t1.depth, t2.depth);
304                 exit(-1);
305             }
306             else if(t2.atomic == A_UNKNOWN && t1.atomic != A_UNKNOWN && t2.depth > t1.depth) {
307                 printf("%s", msg_error);
308                 fprintf(stderr, "Error: incompatible list types; expected depth <= %d instead of %d \n",
309                     t1.depth, t2.depth);
310                 exit(-1);
311             }
312             else if(t1.atomic == A_UNKNOWN && t2.atomic != A_UNKNOWN && t1.depth > t2.depth) {
313                 printf("%s", msg_error);
314                 fprintf(stderr, "Error: incompatible list types; expected something <= %d instead of %d as depth \n",
315                     t2.depth, t1.depth);
316                 exit(-1);
317             }
318             else if(t1.atomic != A_UNKNOWN && t2.atomic != A_UNKNOWN && t1.atomic != t2.atomic) {
319                 printf("%s", msg_error);
320                 fprintf(stderr, "Error: incompatible atomic list types; must be all %s or %s \n",
321                     describeAtomic(t1.atomic), describeAtomic(t2.atomic));
322                 exit(-1);
323             }
324         }
325     void builtinFuncStruct(char *name, Type tipo) {
326         Type param = {D_LIST, MENO_UNO, A_UNKNOWN};
327         int i = binarySearch(TOT, hash(name, TOT));
328         whereInsertFunc(symbolTable, i, name, FUNC, tipo, UNO);
329         int local_i = binarySearch(TOT_FUNC, hash(name, TOT_FUNC));
330         whereInsert(symbolTable[i].local, local_i, "param1", PARAM, param);
331         symbolTable[i].formals.params[0] = symbolTable[i].local[local_i];
332     }

```

```

334 void insertBuiltinFunc() {
335     Type e_tipo = {D_BOOL, 0, A_UNKNOWN};
336     builtinFuncStruct(EMPTY, e_tipo);
337
338     Type l_tipo = {D_INT, 0, A_UNKNOWN};
339     builtinFuncStruct(LENGTH, l_tipo);
340
341     Type htli_tipo = {D_LIST, UNO, A_UNKNOWN};
342     builtinFuncStruct(HEAD, htli_tipo);
343
344     builtinFuncStruct(TAIL, htli_tipo);
345
346     builtinFuncStruct(LAST, htli_tipo);
347
348     builtinFuncStruct(INIT, htli_tipo);
349 }
350
351 int isBuiltinFunc(char *name) {
352     if(strcmp(name,EMPTY)==0) {
353         return TRUE;
354     }
355     else if(strcmp(name,LENGTH)==0) {
356         return TRUE;
357     }
358     else if(strcmp(name,HEAD)==0) {
359         return TRUE;
360     }
361     else if(strcmp(name,TAIL)==0) {
362         return TRUE;
363     }
364     else if(strcmp(name,LAST)==0) {
365         return TRUE;
366     }
367     else if(strcmp(name,INIT)==0) {
368         return TRUE;
369     }
370     return FALSE;
371 }
372
373 void checkBuiltinFunc(char *name, Type p_tipo) {
374     Type expected_p_tipo;
375     int i;
376     if(strcmp(name,EMPTY)==0) {
377         i = binarySearch(TOT, hash(name, TOT));
378         expected_p_tipo = symbolTable[i].formals.params[0].tipo;
379         domainEqual(expected_p_tipo.domain, p_tipo.domain);
380     }
381     else if(strcmp(name,LENGTH)==0) {
382         i = binarySearch(TOT, hash(name, TOT));
383         expected_p_tipo = symbolTable[i].formals.params[0].tipo;
384         domainEqual(expected_p_tipo.domain, p_tipo.domain);
385     }
386     else if(strcmp(name,HEAD)==0 || strcmp(name,LAST)==0) {
387         i = binarySearch(TOT, hash(name, TOT));
388         expected_p_tipo = symbolTable[i].formals.params[0].tipo;
389         domainEqual(expected_p_tipo.domain, p_tipo.domain);
390         if(p_tipo.depth == UNO) {
391             symbolTable[i].tipo.domain = fromAtomicToDomain(p_tipo.atomic);
392             symbolTable[i].tipo.depth = 0;
393             symbolTable[i].tipo.atomic = A_UNKNOWN;
394         }
395         else {
396             symbolTable[i].tipo = p_tipo;
397             symbolTable[i].tipo.depth--;
398         }
399     }
400     else if(strcmp(name,TAIL)==0 || strcmp(name,INIT)==0) {
401         i = binarySearch(TOT, hash(name, TOT));

```

```

402     |     expected_p_tipo = symbolTable[i].formals.params[0].tipo;
403     |     domainEqual(expected_p_tipo.domain, p_tipo.domain);
404     |     symbolTable[i].tipo = p_tipo;
405   }
406 }
407
408 Type exprType(Pnode expr);
409
410 Type idVarType(char *id) {
411     Type t;
412     if(lookup(id, symbolTable, TOT) == TRUE) {
413         int h = binarySearch(TOT, hash(id, TOT));
414         if(strcmp(symbolTable[h].name, id) == 0) {
415             t = symbolTable[h].tipo;
416         }
417         else {
418             t = symbolTable[h].next->tipo;
419         }
420     }
421     else {
422         printf("%s", msg_error);
423         fprintf(stderr, "Error: identifier \"%s\" is an undeclared variable\n", id);
424         exit(-1);
425     }
426     return t;
427 }
428
429 Type defineIdParamType(SymbolTable st, char *id) {
430     if(strcmp(st.name, id) == 0) {
431         return st.tipo;
432     }
433     return st.next->tipo;
434 }
435
436 Type idParamType(char *id) {
437     Type t;
438     int func_pos = funcIndex[funcOid - NUM_BUILT_IN - 2];
439     if(lookup(id, symbolTable[func_pos].local, TOT_FUNC)) {
440         int h = binarySearch(TOT_FUNC, hash(id, TOT_FUNC));
441         t = defineIdParamType(symbolTable[func_pos].local[h], id);
442     }
443     else {
444         printf("%s", msg_error);
445         fprintf(stderr, "Error: identifier \"%s\" is an undeclared parameter \n", id);
446         exit(-1);
447     }
448     return t;
449 }
450
451 Type idNextParamType(char *id) {
452     Type t;
453     int func_pos = funcIndex[funcOid - NUM_BUILT_IN - 2];
454     if(lookup(id, symbolTable[func_pos].next->local, TOT_FUNC)) {
455         int h = binarySearch(TOT_FUNC, hash(id, TOT_FUNC));
456         t = defineIdParamType(symbolTable[func_pos].next->local[h], id);
457     }
458     else {
459         printf("%s", msg_error);
460         fprintf(stderr, "Error: identifier \"%s\" is an undeclared parameter \n", id);
461         exit(-1);
462     }
463     return t;
464 }
465
466 Type simpleExprType(Pnode expr) {
467     Type t;
468     if(expr->type == T_ID) {
469         if(checkVar == FALSE) {

```

```

470     |     if(nextFunc == FALSE)
471     |     t = idParamType(expr->value.sval);
472     |     else
473     |     t = idNextParamType(expr->value.sval);
474     }
475     else
476     |     t = idVarType(expr->value.sval);
477   }
478   else
479   |     t = defineType(expr);
480
481   return t;
482 }
483 Type andOrExprType(Pnode expr, Type t1, Type t2) {
484   t1 = exprType(expr->p1);
485   t2 = exprType(expr->p2);
486   if(t1.domain != D_BOOL) {
487     printf("%s", msg_error);
488     fprintf(stderr, "Error: expected a BOOL type and not %s type \n", describeDomain(t1.domain));
489     exit(-1);
490   }
491   if(t2.domain != D_BOOL) {
492     printf("%s", msg_error);
493     fprintf(stderr, "Error: expected a BOOL type and not %s type \n", describeDomain(t2.domain));
494     exit(-1);
495   }
496   return t1;
497 }
498
499 Type eqNeType(Pnode expr, Type t1, Type t2) {
500   t1 = exprType(expr->p1);
501   t2 = exprType(expr->p2);
502   typeEqual(t1, t2);
503   t1.domain = D_BOOL;
504   return t1;
505 }
506
507 Type comparisonOpExprType(Pnode expr, Type t1, Type t2) {
508   t1 = exprType(expr->p1);
509   t2 = exprType(expr->p2);
510   if(t1.domain != D_INT && t1.domain != D_STRING) {
511     printf("%s", msg_error);
512     fprintf(stderr, "Error: expected a INT or STRING type and not %s type \n", describeDomain(t1.domain));
513     exit(-1);
514   }
515   if(t2.domain != D_INT && t2.domain != D_STRING) {
516     printf("%s", msg_error);
517     fprintf(stderr, "Error: expected a INT or STRING type and not %s type \n", describeDomain(t2.domain));
518     exit(-1);
519   }
520   typeEqual(t1, t2);
521   t1.domain = D_BOOL;
522   return t1;
523 }
524
525 Type arithmeticOpExprType(Pnode expr, Type t1, Type t2) {
526   t1 = exprType(expr->p1);
527   t2 = exprType(expr->p2);
528   if(t1.domain != D_INT) {
529     printf("%s", msg_error);
530     fprintf(stderr, "Error: expected a INT type and not %s type \n", describeDomain(t1.domain));
531     exit(-1);
532   }
533   if(t2.domain != D_INT) {
534     printf("%s", msg_error);
535     fprintf(stderr, "Error: expected a INT type and not %s type \n", describeDomain(t2.domain));
536     exit(-1);
537   }

```

```

538     return t1;
539 }
540
541 Type appendExprType(Pnode expr, Type t1, Type t2) {
542     t1 = exprType(expr->p1);
543     t2 = exprType(expr->p2);
544     if(t1.domain != D_LIST) {
545         printf("%s", msg_error);
546         fprintf(stderr, "Error: expected a LIST type and not %s type \n", describeDomain(t1.domain));
547         exit(-1);
548     }
549     if(t2.domain != D_LIST) {
550         printf("%s", msg_error);
551         fprintf(stderr, "Error: expected a LIST type and not %s type \n", describeDomain(t2.domain));
552         exit(-1);
553     }
554     typeEqual(t1, t2);
555     return t1;
556 }
557
558 Type negMinusExprType(Pnode expr, Type t1, Type t2) {
559     t1 = exprType(expr->p1);
560     if(t1.domain != D_INT) {
561         printf("%s", msg_error);
562         fprintf(stderr, "Error: expected a INT type and not %s type \n", describeDomain(t1.domain));
563         exit(-1);
564     }
565     return t1;
566 }
567
568 Type notExprType(Pnode expr, Type t1, Type t2) {
569     t1 = exprType(expr->p1);
570     if(t1.domain != D_BOOL) {
571         printf("%s", msg_error);
572         fprintf(stderr, "Error: expected a BOOL type and not %s type \n", describeDomain(t1.domain));
573         exit(-1);
574     }
575     return t1;
576 }
577
578 Type rangeExprType(Pnode expr, Type t1, Type t2) {
579     t1 = exprType(expr->p1);
580     t2 = exprType(expr->p2);
581     if(t1.domain != D_INT) {
582         printf("%s", msg_error);
583         fprintf(stderr, "Error: expected a INT type and not %s type \n", describeDomain(t1.domain));
584         exit(-1);
585     }
586     if(t2.domain != D_INT) {
587         printf("%s", msg_error);
588         fprintf(stderr, "Error: expected a INT type and not %s type \n", describeDomain(t2.domain));
589         exit(-1);
590     }
591     t1.domain = D_LIST;
592     t1.depth = 1;
593     t1.atomic = A_INT;
594     return t1;
595 }
596
597 Type ifExprType(Pnode expr, Type t1, Type t2) {
598     Pnode if_node = expr->p1;
599     t1 = exprType(if_node);
600     if(t1.domain != D_BOOL) {
601         printf("%s", msg_error);
602         fprintf(stderr, "Error: expected a BOOL type and not %s type \n", describeDomain(t1.domain));
603         exit(-1);
604     }
605     if_node = if_node->p3;

```

```

606     t1 = exprType(if_node);
607     if_node = if_node->p3;
608     t2 = exprType(if_node);
609     typeEqual(t1, t2);
610     return t1;
611 }
612
613 Type guardExprType(Pnode expr, Type t1, Type t2) {
614     Pnode cond = expr->p1;
615     t1 = exprType(cond);
616     if(t1.domain != D_BOOL) {
617         printf("%s", msg_error);
618         fprintf(stderr, "Error: expected a BOOL type and not %s type \n", describeDomain(t1.domain));
619         exit(-1);
620     }
621     t1 = exprType(expr->p2);
622     if(expr->p3->value.ival == NGUARD_LAST) {
623         t2 = exprType(expr->p3->p1);
624         typeEqual(t1, t2);
625         return t1;
626     }
627     t2 = guardExprType(expr->p3, t1, t1);
628     typeEqual(t1, t2);
629     return t1;
630 }
631
632 Type defineFuncCallExprType(SymbolTable st, char *id, Pnode expr, Type t1, Type t2) {
633     Pnode func_call = expr->p1;
634     if(st.classe != FUNC) {
635         printf("%s", msg_error);
636         fprintf(stderr, "Error: identifier \'%s\' is not a function \n", id);
637         exit(-1);
638     }
639     int indexParams = 0;
640     int builtin = isBuiltinFunc(id);
641     for(func_call = expr->p2->p1; func_call != NULL; func_call = func_call->p3) {
642         if(builtin == TRUE) {
643             checkBuiltinFunc(id, exprType(func_call));
644             int i = binarySearch(TOT, hash(id, TOT));
645             st.tipo = symbolTable[i].tipo;
646             indexParams++;
647         }
648         else {
649             if(checkVar == FALSE) {
650                 sprintf(msg_error, "In the function %s: \n", id);
651             }
652             t1 = exprType(func_call);
653             if(indexParams+1 > st.formals.num) {
654                 indexParams++;
655             }
656             else {
657                 t2 = st.formals.params[indexParams++].tipo;
658                 typeEqual(t1, t2);
659             }
660         }
661     }
662     if(st.formals.num != indexParams) {
663         printf("%s", msg_error);
664         fprintf(stderr, "Error: number of parameters expected for the function \'%s\' is %d and not %d \n",
665                 id, st.formals.num, indexParams);
666         exit(-1);
667     }
668     t1 = st.tipo;
669     return t1;
670 }
671
672 Type funcCallExprType(Pnode expr, Type t1, Type t2) {
673     Pnode func_call = expr->p1;

```

```

674     char *id = func_call->value.sval;
675     int i = binarySearch(TOT, hash(id, TOT));
676     if(lookup(id, symbolTable, TOT) == FALSE) {
677         printf("%s", msg_error);
678         fprintf(stderr, "Error: identifier \"%s\" is an undeclared function \n", id);
679         exit(-1);
680     }
681     if(strcmp(symbolTable[i].name, id) == 0) {
682         return defineFuncCallExprType(symbolTable[i], id, expr, t1, t2);
683     }
684     return defineFuncCallExprType(*symbolTable[i].next, id, expr, t1, t2);
685 }
686
687 Type optionalExprListExprType(Pnode expr, Type t1, Type t2) {
688     Pnode opt_list = expr->p1;
689     Type opt_list_type [TOT_FUNC];
690     int j = 0;
691     t1.domain = D_LIST;
692     int arg_list_depth [TOT_FUNC];
693
694     if(opt_list == NULL) {
695         t1.atomic = A_UNKNOWN;
696         t1.depth = 1;
697         return t1;
698     } else {
699         for(opt_list; opt_list != NULL; opt_list = opt_list->p3) {
700             opt_list_type[j++] = exprType(opt_list);
701         }
702         opt_list_type[j].depth = MENO_UNO;
703
704         for(j=0; j < TOT_FUNC && opt_list_type[j].depth != MENO_UNO; j++) {
705             for(int i=j+1; i < TOT_FUNC && opt_list_type[i].depth != MENO_UNO; i++) {
706                 typeEqual(opt_list_type[j], opt_list_type[i]);
707             }
708             arg_list_depth[j] = opt_list_type[j].depth;
709         }
710         arg_list_depth[j] = MENO_UNO;
711     }
712     for(j=0; j < TOT_FUNC && opt_list_type[j].depth != MENO_UNO; j++) {
713         if(opt_list_type[j].domain == D_LIST) {
714             t1.atomic = opt_list_type[j].atomic;
715             if(t1.atomic != A_UNKNOWN)
716                 break;
717         } else {
718             t1.atomic = fromDomainToAtomic(opt_list_type[j].domain);
719             break;
720         }
721     }
722     t1.depth = maxArray(arg_list_depth) + 1;
723     return t1;
724 }
725
726
727 Type assignStatExprType(Pnode expr, Type t1, Type t2) {
728     char *id = expr->p1->value.sval;
729     if(lookup(id, symbolTable, TOT) == FALSE) {
730         printf("%s", msg_error);
731         fprintf(stderr, "Error: identifier \"%s\" is an undeclared variable \n", id);
732         exit(-1);
733     }
734     int index = binarySearch(TOT, hash(id, TOT));
735     if(strcmp(symbolTable[index].name, id) == 0) {
736         t1 = symbolTable[index].tipo;
737     }
738     else {
739         t1 = symbolTable[index].next->tipo;
740     }
741     t2 = exprType(expr->p1->p3);

```

```

742     typeEqual(t1,t2);
743     return t1;
744 }
745
746 Type exprType(Pnode expr) {
747     Type t1;
748     Type t2;
749     if(expr->type != T_NONTERMINAL) {
750         return simpleExprType(expr);
751     }
752     else if(expr->value.ival == NAND || expr->value.ival == NOR) {
753         return andOrExprType(expr, t1, t2);
754     }
755     else if(expr->value.ival == NEQ || expr->value.ival == NNE) {
756         return eqNeType(expr, t1, t2);
757     }
758     else if(expr->value.ival == NGR || expr->value.ival == NGE || expr->value.ival == NLS
759             || expr->value.ival == NLE) {
760         return comparisonOpExprType(expr, t1, t2);
761     }
762     else if(expr->value.ival == NPLUS || expr->value.ival == NMINUS || expr->value.ival == NMULT
763             || expr->value.ival == NDIV) {
764         return arithmeticOpExprType(expr, t1, t2);
765     }
766     else if(expr->value.ival == NAPP) {
767         return appendExprType(expr, t1, t2);
768     }
769     else if(expr->value.ival == NNEG_MINUS) {
770         return negMinusExprType(expr, t1, t2);
771     }
772     else if(expr->value.ival == NNOT) {
773         return notExprType(expr, t1, t2);
774     }
775     else if(expr->value.ival == NRANGE) {
776         return rangeExprType(expr, t1, t2);
777     }
778     else if(expr->value.ival == NIF_EXPR) {
779         return ifExprType(expr, t1, t2);
780     }
781     else if(expr->value.ival == NGUARD_EXPR) {
782         return guardExprType(expr, t1, t2);
783     }
784     else if(expr->value.ival == NFUNC_CALL) {
785         return funcCallExprType(expr, t1, t2);
786     }
787     else if(expr->value.ival == NINPUT) {
788         return defineType(expr->p1->p1);
789     }
790     else if(expr->value.ival == NOUPUT || expr->value.ival == NSHOW_STAT) {
791         return exprType(expr->p1);
792     }
793     else if(expr->value.ival == NOPT_EXPR_LIST) {
794         return optionalExprListExprType(expr, t1, t2);
795     }
796     else if(expr->value.ival == NASSIGN_STAT) {
797         return assignStatExprType(expr, t1, t2);
798     }
799     else {
800         error("Unknown yacc instruction \n");
801     }
802     return t1;
803 }
804
805 void insertVariable(Pnode pnode) {
806     for(Pnode p = pnode->p1; p != NULL; p = p->p3) {
807         Pnode var = p->p1->p1;
808         Pnode t = p->p1->p3;
809         for(var; var != NULL; var = var->p3) {

```

```

810     |     |     char *name = var->value.sval;
811     |     |     if(lookup(name, symbolTable, TOT) == FALSE) {
812     |     |     |     Type tipo = defineType(t->p1);
813     |     |     |     int index = binarySearch(TOT, hash(name, TOT));
814     |     |     |     whereInsert(symbolTable, index, name, VAR, tipo);
815     |     |     }
816     |     |     else {
817     |     |     |     fprintf(stderr, "Error: identifier \"%s\" is a duplicate variable \n", name);
818     |     |     |     exit(-1);
819     |     |     }
820   }
821 }
823
824 int insertFunction(Pnode f, Pnode param) {
825     int index;
826     char *name = f->value.sval;
827     msg_error = malloc(sizeof(msg_error) + OFFSET);
828     sprintf(msg_error, "In the function %s: \n", name);
829     Pnode type = f->p3->p3->p1;
830     if(lookup(name, symbolTable, TOT) == FALSE) {
831         Type tipo = defineType(type);
832         index = binarySearch(TOT, hash(name, TOT));
833         int numParameters = numParam(param);
834         whereInsertFunc(symbolTable, index, name, FUNC, tipo, numParameters);
835     }
836     else {
837         fprintf(stderr, "Error: identifier \"%s\" is a duplicate function or it has been declared as a variable \n",
838                 |     |     |     name);
839         exit(-1);
840     }
841     return index;
842 }
843
844 void insertParameter(SymbolTable st, Pnode param, int j) {
845     char *name = param->p1->value.sval;
846     if(lookup(name, st.local, TOT_FUNC) == FALSE) {
847         Type tipo = defineType(param->p2->p1);
848         int local_index = binarySearch(TOT_FUNC, hash(name, TOT_FUNC));
849         whereInsert(st.local, local_index, name, PARAM, tipo);
850
851         if(strcmp(st.local[local_index].next->name, NOTHING) != 0) {
852             st.formals.params[j] = *st.local[local_index].next;
853         } else {
854             st.formals.params[j] = st.local[local_index];
855         }
856         paramOid++;
857     }
858     else {
859         printf("%s", msg_error);
860         fprintf(stderr, "Error: identifier \"%s\" is a duplicate parameter \n", name);
861         exit(-1);
862     }
863 }
864
865 int createAndCheckSymbolTable() {
866     Pnode pnode;
867     char *id;
868     int func_pos = 0;
869     insertBuiltinFunc();
870
871     pnode = root->p1;
872     insertVariable(pnode);
873
874     pnode = pnode->p3;
875     for(Pnode p = pnode->p1; p != NULL; p = p->p3) {
876         Pnode f = p->p1;
877         Pnode param = f->p3->p1;

```

```

878     int j = 0;
879     funcIndex[func_pos] = insertFunction(f, param);
880     for(param; param != NULL; param = param->p3) {
881         char *id = f->value.sval;
882         int h = binarySearch(TOT_FUNC, hash(id, TOT_FUNC));
883         if(strcmp(symbolTable[funcIndex[func_pos]].name, id) == 0) {
884             insertParameter(symbolTable[funcIndex[func_pos]], param, j++);
885         }
886         else {
887             insertParameter(*symbolTable[funcIndex[func_pos]].next, param, j++);
888         }
889     }
890     func_pos++;
891     paramOid = 0;
892 }
893
894 func_pos = 0;
895 funcOid = NUM_BUILT_IN + 1;
896 for(Pnode p = pnode->p1; p != NULL; p = p->p3) {
897     funcOid++;
898     char *id = p->p1->value.sval;
899     int h = binarySearch(TOT_FUNC, hash(id, TOT_FUNC));
900     msg_error = malloc(sizeof(msg_error) + OFFSET);
901     sprintf(msg_error, "In the function %s: \n", id);
902     Type fType;
903     if(strcmp(symbolTable[funcIndex[func_pos]].name, id) == 0) {
904         fType = symbolTable[funcIndex[func_pos++]].tipo;
905     }
906     else {
907         nextFunc = TRUE;
908         fType = symbolTable[funcIndex[func_pos++]].next->tipo;
909     }
910     Pnode expr = p->p1->p3->p3->p3;
911     Type eType = exprType(expr);
912     typeEqual(fType, eType);
913     nextFunc = FALSE;
914 }
915
916 pnode = pnode->p3;
917 checkVar = TRUE;
918 int count = 1;
919 for(Pnode body = pnode->p1; body != NULL; body = body->p3) {
920     msg_error = malloc(sizeof(msg_error) + OFFSET);
921     sprintf(msg_error, "In the statement number %d: \n", count++);
922     Type eType = exprType(body);
923 }
924 return 0;
925 }
926
927 void printElement(FILE *fp, SymbolTable st, int i, int indent) {
928     if(isBuiltinFunc(st.name)) {
929         fprintf(fp, "Nome: %s, Classe: %s, Oid:%d, Numero_Parametri: %d \n\n",
930                 | st.name, describeClass(st.classe), st.oid, st.formals.num);
931         if(strcmp(st.next->name, NOTHING) != 0) {
932             indent++;
933             for(int j=0; j<indent; j++) {
934                 fprintf(fp, "%s", " ");
935             }
936             fprintf(fp, "Descrittore collegato al posto %d della Symbol Table\n",i);
937             printElement(fp, *st.next, i, indent);
938         }
939         return;
940     }
941     for(int j=0; j<indent; j++) {
942         fprintf(fp, "%s", " ");
943     }
944     fprintf(fp, "Nome: %s, Classe: %s, Tipo: (Dominio: %s, Profondita: %d, Tipo_Atomico: %s), Oid:%d",
945             | st.name, describeClass(st.classe),

```

```

946     describeDomain(st.tipo.domain), st.tipo.depth, describeAtomic(st.tipo.atomic), st.oid);
947     if(st.classe == FUNC) {
948         fprintf(fp, ", Numero_Parametri: %d \n\n", st.formals.num);
949         indent++;
950         for(int j=0; j<indent; j++) {
951             fprintf(fp, "%s", "    ");
952         }
953         fprintf(fp, "%s", "Parametri della funzione: \n\n");
954         if(st.formals.num == 0) {
955             for(int j=0; j<indent; j++) {
956                 fprintf(fp, "%s", "    ");
957             }
958             fprintf(fp, "%s", "nessuno\n\n");
959         } else {
960             for(int i=0; i < TOT_FUNC; i++) {
961                 if(st.local[i].name != NULL) {
962                     for(int j=0; j<indent; j++) {
963                         fprintf(fp, "%s", "    ");
964                     }
965                     fprintf(fp, "Contenuto della Local Symbol Table nel posto %d \n", i);
966                     printElement(fp, st.local[i], i, indent);
967                 }
968             }
969         }
970     } else {
971         fprintf(fp, "%s", "\n\n");
972     }
973     if(strcmp(st.next->name, NOTHING) != 0) {
974         indent++;
975         for(int j=0; j<indent; j++) {
976             fprintf(fp, "%s", "    ");
977         }
978         if(st.classe == PARAM) {
979             fprintf(fp, "Descrittore collegato al posto %d della Local Symbol Table\n", i);
980             printElement(fp, *st.next, i, indent);
981         }
982         else {
983             fprintf(fp, "Descrittore collegato al posto %d della Symbol Table\n", i);
984             printElement(fp, *st.next, i, indent);
985         }
986     }
987 }
988
989 void semantics() {
990     parser();
991     createAndCheckSymbolTable();
992
993     FILE *fp;
994     fp = fopen("SymbolTable.txt", "w");
995
996     for(int i=0; i < TOT; i++) {
997         if(symbolTable[i].name != NULL) {
998             fprintf(fp, "Contenuto Symbol Table nel posto %d \n", i);
999             printElement(fp, symbolTable[i], i, 0);
1000         }
1001     }
1002     fclose(fp);
1003 }
1004

```

Appendice D

Il codice C completo del generatore di codice intermedio di Tofu, T-code (file *gen.c*).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "semantics.c"
5
6 #define ARGS_DIM 2
7 #define FUNC_DIM 100
8 #define FIRST_DECL_FUNC 7
9
10 typedef struct statement{
11     int address;
12     Operator op;
13     Value args [ARGS_DIM];
14     struct statement *next;
15 } Stat;
16
17 typedef struct {
18     Stat *head;
19     int size;
20     Stat *tail;
21 } Code;
22
23 int func_index;
24 int var_oid;
25 int entry[FUNC_DIM];
26 int verify_var = FALSE;
27 int check_builtin = FALSE;
28
29 void relocate(Code code, int offset) {
30     Stat *pt = code.head;
31
32     for(int i=0; i < code.size; i++) {
33         pt->address += offset;
34         pt = pt->next;
35     }
36 }
37
38 Code appcode(Code code1, Code code2) {
39     Code rescode;
40
41     relocate(code2, code1.size);
42     rescode.head = code1.head;
43     rescode.tail = code2.tail;
44     code1.tail->next = code2.head;
45     rescode.size = code1.size + code2.size;
46     return rescode;
47 }
48
49 Code endcode() {
50     static Code code = {NULL, 0, NULL};
51     return code;
52 }
53
54 Code concode(Code code1, Code code2, ...) {
55     Code rescode = code1;
56     Code *PCODE = &code2;
57     while(pcode->head != NULL) {
```

```

58     |     rescode = appcode(rescode, *PCODE);
59     |     PPCODE++;
60   }
61   return rescode;
62 }
63
64 Stat *newstat(Operator op) {
65     Stat *pstat;
66     pstat = (Stat*) malloc(sizeof(stat));
67     pstat->address = 0;
68     pstat->op = op;
69     pstat->next = NULL;
70     return pstat;
71 }
72
73 Code makecode(Operator op) {
74     Code code;
75     code.head = code.tail = newstat(op);
76     code.size = UNO;
77     return code;
78 }
79
80 Code makecode1(Operator op, int arg) {
81     Code code;
82     code = makecode(op);
83     code.head->args[0].ival = arg;
84     return code;
85 }
86
87 Code makecode2(Operator op, int arg1, int arg2) {
88     Code code;
89     code = makecode(op);
90     code.head->args[0].ival = arg1;
91     code.head->args[1].ival = arg2;
92     return code;
93 }
94
95 Code make_vars(int num_var) {
96     return makecode1(VARS, num_var);
97 }
98
99 Code make_halt() {
100    return makecode(HALT);
101 }
102
103 Code make_newi() {
104    return makecode(NEWI);
105 }
106
107 Code make_news() {
108    return makecode(NEWS);
109 }
110
111 Code make_newb() {
112    return makecode(NEWB);
113 }
114
115 Code make_newl() {
116    return makecode(NEWL);
117 }
118
119 Code make_loci(int num) {
120    return makecode1(LOCI, num);
121 }
122
123 Code make_locb(int b) {
124    return makecode1(LOCB, b);
125 }

```

```

126
127 Code make_locs(char *s) {
128     Code code;
129     code = makecode(LOCS);
130     code.head->args[0].sval = s;
131     return code;
132 }
133
134 Code make_list(int size) {
135     return makecode1(LIST, size);
136 }
137
138 Code make_load(int flag, int oid) {
139     return makecode2(LOAD, flag, oid);
140 }
141
142 Code make_stor(int oid) {
143     return makecode1(STOR, oid);
144 }
145
146 Code make_and(int offset, Code code, int skip, int f_loci) {
147     return concode(makecode1(SKPF, offset), code, makecode1(SKIP, skip), make_locb(f_loci), endcode());
148 }
149
150 Code make_or(int skpf, int t_loci, int exit) {
151     return concode(makecode1(SKPF, skpf), make_locb(t_loci), makecode1(SKIP, exit), endcode());
152 }
153
154 Code make_equa() {
155     return makecode(EQUA);
156 }
157
158 Code make_nequ() {
159     return makecode(NEQU);
160 }
161
162 Code make_grtr() {
163     return makecode(GRTR);
164 }
165
166 Code make_greq() {
167     return makecode(GREQ);
168 }
169
170 Code make_leth() {
171     return makecode(LETH);
172 }
173
174 Code make_leeq() {
175     return makecode(LEEEQ);
176 }
177
178 Code make_plus() {
179     return makecode(PPLUS);
180 }
181
182 Code make_minu() {
183     return makecode(MINU);
184 }
185
186 Code make_mult() {
187     return makecode(MULT);
188 }
189
190 Code make_divi() {
191     return makecode(DIVI);
192 }
193

```

```

194  Code make_appn() {
195  |   return makecode(APPN);
196  }
197
198  Code make_umin() {
199  |   return makecode(UMIN);
200  }
201
202  Code make_nega() {
203  |   return makecode(NEGA);
204  }
205
206  Code make_rang() {
207  |   return makecode(RANG);
208  }
209
210  Code make_empt() {
211  |   return makecode(T_EMPT);
212  }
213
214  Code make_leng() {
215  |   return makecode(T LENG);
216  }
217
218  Code make_head() {
219  |   return makecode(T HEAD);
220  }
221
222  Code make_tail() {
223  |   return makecode(T TAIL);
224  }
225
226  Code make_last() {
227  |   return makecode(T LAST);
228  }
229
230  Code make_init() {
231  |   return makecode(T INIT);
232  }
233
234  Code make_func_call(int numparams, int entry) {
235  |   return concode(makecode1(PUSH, numparams), makecode1(JUMP, entry), makecode(APOP), endcode());
236  }
237
238  Code make_if_expr(int offset, Code code, int exit) {
239  |   return concode(makecode1(SKPF, offset), code, makecode1(SKIP, exit), endcode());
240  }
241
242  Code make_guard_expr(int offset, Code code, int exit) {
243  |   return concode(makecode1(SKPF, offset), code, makecode1(SKIP, exit), endcode());
244  }
245
246  Code make_func_decl(int fid, Code code) {
247  |   return concode(makecode1(T FUNC, fid), code, makecode(RETN), endcode());
248  }
249
250  Code make_geti() {
251  |   return makecode(GETI);
252  }
253
254  Code make_gets() {
255  |   return makecode(GETS);
256  }
257
258  Code make_getb() {
259  |   return makecode(GETB);
260  }
261

```

```

262     Code make_getl(int depth, int atomic) {
263         return makecode2(GETL, depth, atomic);
264     }
265
266     Code make_puts() {
267         return makecode(PUTS);
268     }
269
270     Code make_show() {
271         return makecode(T_SHOW);
272     }
273
274     Code genVarCode();
275
276     Code genVarIntCode(Code code) {
277         var_oid++;
278         code = genVarCode();
279         if(code.size != 0) {
280             return concode(make_newi(), code, endcode());
281         }
282         return make_newi();
283     }
284
285     Code genVarStringCode(Code code) {
286         var_oid++;
287         code = genVarCode();
288         if(code.size != 0) {
289             return concode(make_news(), code, endcode());
290         }
291         return make_news();
292     }
293
294     Code genVarBoolCode(Code code) {
295         var_oid++;
296         code = genVarCode();
297         if(code.size != 0) {
298             return concode(make_newb(), code, endcode());
299         }
300         return make_newb();
301     }
302
303     Code genVarListCode(Code code) {
304         var_oid++;
305         code = genVarCode();
306         if(code.size != 0) {
307             return concode(make_newl(), code, endcode());
308         }
309         return make_newl();
310     }
311
312     Code genVarCode() {
313         Code code = endcode();
314         for(int i=0; i < TOT; i++) {
315             if(symbolTable[i].name != NULL && symbolTable[i].classe == VAR && symbolTable[i].oid == var_oid) {
316                 switch(symbolTable[i].tipo.domain) {
317                     case D_INT:
318                         return genVarIntCode(code);
319                     case D_STRING:
320                         return genVarStringCode(code);
321                     case D_BOOL:
322                         return genVarBoolCode(code);
323                     case D_LIST:
324                         return genVarListCode(code);
325                 }
326             }
327         }
328         return code;
329     }

```

```

330
331     Code genBuiltinCode(char *name) {
332         Code code;
333         if(strcmp(name, EMPTY)==0) {
334             check_builtin = TRUE;
335             code = make_empty();
336         }
337         else if(strcmp(name, LENGTH)==0) {
338             check_builtin = TRUE;
339             code = make_leng();
340         }
341         else if(strcmp(name, HEAD)==0) {
342             check_builtin = TRUE;
343             code = make_head();
344         }
345         else if(strcmp(name, TAIL)==0) {
346             check_builtin = TRUE;
347             code = make_tail();
348         }
349         else if(strcmp(name, LAST)==0) {
350             check_builtin = TRUE;
351             code = make_last();
352         }
353         else if(strcmp(name, INIT)==0) {
354             check_builtin = TRUE;
355             code = make_init();
356         }
357         else
358             code = endcode();
359         return code;
360     }
361
362     Code genConstCode(Pnode p) {
363         Code code;
364
365         if(p->type == T_INTCONST)
366             code = make_loci(p->value.ival);
367         else if(p->type == T_STRCONST)
368             code = make_locs(p->value.sval);
369         else if (p->type == T_BOOLCONST)
370             code = make_locb(p->value.ival);
371
372         return code;
373     }
374
375     Code genIdCode(Pnode p) {
376         Code code;
377
378         if(p->type == T_INT)
379             code = make_geti();
380         else if(p->type == T_STRING)
381             code = make_gets();
382         else if (p->type == T_BOOL)
383             code = make_getb();
384         else if (p->value.ival == NLIST_TYPE) {
385             Type t = defineType(p);
386             code = make_getl(t.depth, t.atomic);
387         }
388
389         return code;
390     }
391
392     Code genSimpleExprCode(Pnode expr) {
393         Code code;
394
395         if(expr->type == T_ID) {
396             if(verify_var == FALSE) {
397                 Formals formals = symbolTable[func_index].formals;

```

```

398     |         for(int i=0; i < formals.num; i++) {
399     |             if(strcmp(expr->value.sval, formals.params[i].name)==0) {
400     |                 code = make_load(1, formals.params[i].oid);
401     |             }
402     |         }
403     |     } else {
404     |         int index = binarySearch(TOT, hash(expr->value.sval, TOT));
405     |         code = make_load(0, symbolTable[index].oid);
406     |     }
407     | }
408     |
409     | else {
410     |     code = genConstCode(expr);
411     | }
412     return code;
413 }
414
415 Code genExprCode(Pnode expr);
416
417 Code genIfExprCode(Pnode expr) {
418     Code code_expr2 = genExprCode(expr->p1->p3);
419     Code code_expr3 = genExprCode(expr->p1->p3->p3);
420     return concode(genExprCode(expr->p1), make_if_expr(code_expr2.size + 2, code_expr2, code_expr3.size + 1),
421     |     |     |     |     code_expr3, endcode());
422 }
423
424 Code genGuardExprCode(Pnode expr) {
425     int array_exit[70];
426     int i = 0;
427     Code guard_code = endcode();
428     Pnode tmp_expr = expr;
429     while(tmp_expr->value.ival != NGUARD_LAST) {
430         Code tmp_cond = genExprCode(tmp_expr->p1);
431         Code tmp_guard_expr = genExprCode(tmp_expr->p2);
432         array_exit[i++] = tmp_cond.size + tmp_guard_expr.size + 2;
433         tmp_expr = tmp_expr->p3;
434     }
435     array_exit[i++] = genExprCode(tmp_expr->p1).size + 1;
436     array_exit[i] = -1;
437     for(int k=0; array_exit[k] != -1; k++) {
438     }
439     i = 1;
440     while(expr->value.ival != NGUARD_LAST) {
441         Code cond = genExprCode(expr->p1);
442         Code guard_expr = genExprCode(expr->p2);
443         int exit = 0;
444         for(int j = i; array_exit[j] != -1; j++) {
445             exit = exit + array_exit[j];
446         }
447         if(guard_code.size != 0) {
448             guard_code = concode(guard_code, cond, make_guard_expr(guard_expr.size + 2, guard_expr, exit), endcode());
449         }
450         else {
451             guard_code = concode(cond, make_guard_expr(guard_expr.size + 2, guard_expr, exit), endcode());
452         }
453         i++;
454         expr = expr->p3;
455     }
456     return concode(guard_code, genExprCode(expr->p1), endcode());
457 }
458
459 Code genFuncCallExprCode(Pnode expr, Code code) {
460     char *id = expr->p1->value.sval;
461     Pnode func_call = expr->p2->p1;
462     int num_param = 0;
463     int index = binarySearch(TOT, hash(id, TOT));
464     int jump = entry[symbolTable[index].oid];
465

```

```

466     if(func_call != NULL) {
467         Code param_code = genExprCode(func_call);
468         num_param++;
469
470         code = genBuiltInCode(id);
471         if(check_builtin == TRUE) {
472             code = concode(param_code, code, endcode());
473         }
474         else {
475             func_call = func_call->p3;
476             for(func_call; func_call != NULL; func_call = func_call->p3) {
477                 param_code = concode(param_code, genExprCode(func_call), endcode());
478                 num_param++;
479             }
480             code = concode(param_code, make_func_call(num_param, jump), endcode());
481         }
482     }
483     else {
484         code = make_func_call(num_param, jump);
485     }
486     check_builtin = FALSE;
487     return code;
488 }
489
490 Code genOptionalExprListExprCode(Pnode expr) {
491     int size = 0;
492     if(expr->p1 != NULL) {
493         Code list_code = genExprCode(expr->p1);
494         size++;
495         Pnode opt_list = expr->p1->p3;
496
497         for(opt_list; opt_list != NULL; opt_list = opt_list->p3) {
498             list_code = concode(list_code, genExprCode(opt_list), endcode());
499             size++;
500         }
501         return concode(list_code, make_list(size), endcode());
502     }
503     return make_list(size);
504 }
505
506 Code genAssignStatExprCode(Pnode expr) {
507     char *id = expr->p1->value.sval;
508     int index = binarySearch(TOT, hash(id, TOT));
509     return concode(genExprCode(expr->p1->p3), make_stor(symbolTable[index].oid), endcode());
510 }
511
512 Code genExprCode(Pnode expr) {
513     Code code;
514     if(expr->type != T_NONTERMINAL) {
515         return genSimpleExprCode(expr);
516     }
517     else if(expr->value.ival == NAND) {
518         Code code_expr2 = genExprCode(expr->p2);
519         return concode(genExprCode(expr->p1), make_and(code_expr2.size + 2, code_expr2, 2, 0), endcode());
520     }
521     else if(expr->value.ival == NOR) {
522         Code code_expr2 = genExprCode(expr->p2);
523         return concode(genExprCode(expr->p1), make_or(3, 1, code_expr2.size + 1), code_expr2, endcode());
524     }
525     else if(expr->value.ival == NEQ) {
526         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_equa(), endcode());
527     }
528     else if(expr->value.ival == NNE) {
529         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_nequ(), endcode());
530     }
531     else if(expr->value.ival == NGR) {
532         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_grtr(), endcode());
533     }

```

```

534     else if(expr->value.ival == NGE) {
535         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_greq(), endcode());
536     }
537     else if(expr->value.ival == NLS) {
538         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_leth(), endcode());
539     }
540     else if(expr->value.ival == NLE) {
541         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_leeq(), endcode());
542     }
543     else if(expr->value.ival == NPPLUS) {
544         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_plus(), endcode());
545     }
546     else if(expr->value.ival == NMINUS) {
547         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_minu(), endcode());
548     }
549     else if(expr->value.ival == NMULT) {
550         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_mult(), endcode());
551     }
552     else if(expr->value.ival == NDIV) {
553         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_divi(), endcode());
554     }
555     else if(expr->value.ival == NAPP) {
556         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_appn(), endcode());
557     }
558     else if(expr->value.ival == NNEG_MINUS) {
559         return concode(genExprCode(expr->p1), make_umin(), endcode());
560     }
561     else if(expr->value.ival == NNOT) {
562         return concode(genExprCode(expr->p1), make_nega(), endcode());
563     }
564     else if(expr->value.ival == NRANGE) {
565         return concode(genExprCode(expr->p1), genExprCode(expr->p2), make_rang(), endcode());
566     }
567     else if(expr->value.ival == NIF_EXPR) {
568         return genIfExprCode(expr);
569     }
570     else if(expr->value.ival == NGUARD_EXPR) {
571         return genGuardExprCode(expr);
572     }
573     else if(expr->value.ival == NFUNC_CALL) {
574         return genFuncCallExprCode(expr, code);
575     }
576     else if(expr->value.ival == NINPUT) {
577         return genIdCode(expr->p1->p1);
578     }
579     else if(expr->value.ival == NOUTPUT) {
580         return concode(genExprCode(expr->p1), make_puts(), endcode());
581     }
582     else if(expr->value.ival == NSHOW_STAT) {
583         return concode(genExprCode(expr->p1), make_show(), endcode());
584     }
585     else if(expr->value.ival == NOPT_EXPR_LIST) {
586         return genOptionalExprListExprCode(expr);
587     }
588     else if(expr->value.ival == NASSIGN_STAT) {
589         return genAssignStatExprCode(expr);
590     }
591     else {
592         error("Unknown yacc instruction \n");
593     }
594     return code;
595 }
596
597 void *TcodeString(FILE *fp, Operator op, Value arg1, Value arg2) {
598     switch(op) {
599         case VARS:
600             fprintf(fp, "VARS %d\n", arg1.ival);
601             break;

```

```

602     case HALT:
603         fprintf(fp, "HALT\n");
604         break;
605     case NEWI:
606         fprintf(fp, "NEWI\n");
607         break;
608     case NEWS:
609         fprintf(fp, "NEWS\n");
610         break;
611     case NEWB:
612         fprintf(fp, "NEWB\n");
613         break;
614     case NEWL:
615         fprintf(fp, "NEWL\n");
616         break;
617     case LOCI:
618         fprintf(fp, "LOCI %d\n", arg1.ival);
619         break;
620     case LOCB:
621         fprintf(fp, "LOCB %d\n", arg1.ival);
622         break;
623     case LOCS:
624         fprintf(fp, "LOCS %s\n", arg1.sval);
625         break;
626     case LIST:
627         fprintf(fp, "LIST %d\n", arg1.ival);
628         break;
629     case LOAD:
630         fprintf(fp, "LOAD %d %d\n", arg1.ival, arg2.ival);
631         break;
632     case STOR:
633         fprintf(fp, "STOR %d\n", arg1.ival);
634         break;
635     case SKPF:
636         fprintf(fp, "SKPF %d\n", arg1.ival);
637         break;
638     case SKIP:
639         fprintf(fp, "SKIP %d\n", arg1.ival);
640         break;
641     case EQUA:
642         fprintf(fp, "EQUA\n");
643         break;
644     case NEQU:
645         fprintf(fp, "NEQU\n");
646         break;
647     case GRTR:
648         fprintf(fp, "GRTR\n");
649         break;
650     case GREQ:
651         fprintf(fp, "GREQ\n");
652         break;
653     case LETH:
654         fprintf(fp, "LETH\n");
655         break;
656     case LEEQ:
657         fprintf(fp, "LEEQ\n");
658         break;
659     case PLUS:
660         fprintf(fp, "PLUS\n");
661         break;
662     case MINU:
663         fprintf(fp, "MINU\n");
664         break;
665     case MULT:
666         fprintf(fp, "MULT\n");
667         break;
668     case DIVI:
669         fprintf(fp, "DIVI\n");

```

```

670     break;
671 case APPN:
672     fprintf(fp, "APPN\n");
673     break;
674 case UMIN:
675     fprintf(fp, "UMIN\n");
676     break;
677 case NEGA:
678     fprintf(fp, "NEGA\n");
679     break;
680 case RANG:
681     fprintf(fp, "RANG\n");
682     break;
683 case T_EMPT:
684     fprintf(fp, "EMPT\n");
685     break;
686 case T LENG:
687     fprintf(fp, "LENG\n");
688     break;
689 case T HEAD:
690     fprintf(fp, "HEAD\n");
691     break;
692 case T TAIL:
693     fprintf(fp, "TAIL\n");
694     break;
695 case T LAST:
696     fprintf(fp, "LAST\n");
697     break;
698 case T INIT:
699     fprintf(fp, "INIT\n");
700     break;
701 case PUSH:
702     fprintf(fp, "PUSH %d\n", arg1.ival);
703     break;
704 case JUMP:
705     fprintf(fp, "JUMP %d\n", arg1.ival);
706     break;
707 case APOP:
708     fprintf(fp, "APOP\n");
709     break;
710 case T FUNC:
711     fprintf(fp, "FUNC %d\n", arg1.ival);
712     break;
713 case RETN:
714     fprintf(fp, "RETN\n");
715     break;
716 case GETI:
717     fprintf(fp, "GETI\n");
718     break;
719 case GETS:
720     fprintf(fp, "GETS\n");
721     break;
722 case GETB:
723     fprintf(fp, "GETB\n");
724     break;
725 case GETL:
726     fprintf(fp, "GETL %d %s\n", arg1.ival, describeAtomic(arg2.ival));
727     break;
728 case PUTS:
729     fprintf(fp, "PUTS\n");
730     break;
731 case T SHOW:
732     fprintf(fp, "SHOW\n");
733     break;
734 default: error("Unknown T-code instruction \n");
735     break;
736 }
737 }
```

```

738
739 Code varBodyTcode() {
740     Code var_code = genVarCode();
741     Code code;
742     Pnode pnode = root->p1->p3->p3->p1;
743     verify_var = TRUE;
744     if(var_code.size != 0) {
745         code = concode(make_vars(var_code.size), var_code, genExprCode(pnode), endcode());
746     }
747     else {
748         code = concode(make_vars(var_code.size), genExprCode(pnode), endcode());
749     }
750     for(pnode = pnode->p3; pnode != NULL; pnode = pnode->p3) {
751         code = concode(code, genExprCode(pnode), endcode());
752     }
753     return concode(code, make_halt(), endcode());
754 }
755
756 Code funcTcodeSenzaJump(Pnode p, Code varBody) {
757     Code func_code;
758     int j = FIRST_DECL_FUNC + 1;
759
760     if(p != NULL) {
761         char *name = p->p1->value.sval;
762         func_index = binarySearch(TOT, hash(name, TOT));
763         entry[FIRST_DECL_FUNC] = varBody.tail->address + 1;
764         Pnode expr = p->p1->p3->p3->p3;
765         func_code = make_func_decl(symbolTable[func_index].oid, genExprCode(expr));
766         for(p = p->p3; p != NULL; p = p->p3) {
767             name = p->p1->value.sval;
768             func_index = binarySearch(TOT, hash(name, TOT));
769             entry[j++] = entry[FIRST_DECL_FUNC] + func_code.tail->address + 1;
770             expr = p->p1->p3->p3->p3;
771             func_code = concode(func_code, make_func_decl(symbolTable[func_index].oid, genExprCode(expr)), endcode());
772         }
773         return func_code;
774     }
775
776     return endcode();
777 }
778
779 void genTcode(FILE *fp) {
780     Code code = varBodyTcode();
781     Code func_code;
782     Pnode p = root->p1->p3->p1;
783     Pnode pSenzaJump = p;
784     verify_var = FALSE;
785
786     func_code = funcTcodeSenzaJump(pSenzaJump, code);
787
788     if(p != NULL) {
789         char *name = p->p1->value.sval;
790         func_index = binarySearch(TOT, hash(name, TOT));
791         Pnode expr = p->p1->p3->p3->p3;
792         func_code = make_func_decl(symbolTable[func_index].oid, genExprCode(expr));
793         for(p = p->p3; p != NULL; p = p->p3) {
794             name = p->p1->value.sval;
795             func_index = binarySearch(TOT, hash(name, TOT));
796             expr = p->p1->p3->p3->p3;
797             func_code = concode(func_code, make_func_decl(symbolTable[func_index].oid, genExprCode(expr)), endcode());
798         }
799     }
800     else {
801         func_code = endcode();
802     }
803     var_oid = 0;
804     if(func_code.size != 0) {
805         code = concode(varBodyTcode(), func_code, endcode());

```

```
806     }
807     else {
808         code = varBodyTcode();
809     }
810     fprintf(fp, "TCODE %d\n", code.size);
811     for(Stat *stat = code.head; stat != NULL; stat = stat->next) {
812         TcodeString(fp, stat->op, stat->args[0], stat->args[1]);
813     }
814 }
815
816 int main() {
817     semantics();
818
819     FILE *fp;
820     fp = fopen("IntermediateCode.txt", "w");
821     genTcode(fp);
822     fclose(fp);
823
824     return 0;
825 }
826
```

Appendice E

Il codice C completo della macchina virtuale di Tofu (file *interpreter.c*).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "def.h"
5
6 #define MAX_ARGS 2
7 #define MAX_IP_OBJECTS 30000
8 #define MAX_OBJECTS 5000
9 #define STRING_DIM 150
10 #define TRUE 1
11 #define FALSE 0
12
13 typedef struct {
14     Operator oper;
15     Value args[MAX_ARGS];
16 } Tcode;
17
18 typedef enum {
19     D_INT,
20     D_STRING,
21     D_BOOL,
22     D_LIST
23 } Domain;
24
25 typedef enum {
26     A_UNKNOWN,
27     A_INT,
28     A_STRING,
29     A_BOOL
30 } Atomic;
31
32 typedef struct {
33     Domain type;
34     int leng;
35     Value inst;
36 } Object;
37
38 typedef struct {
39     int num;
40     int objs;
41     int ret;
42 } Aobject;
43
44 typedef struct {
45     char *input;
46 } ArgsMain;
47
48 Tcode *prog;
49 int size;
50 int ap = -1;
51 int pc, vp, op, ip, pp;
52 Object *vars;
53 Object ostack[MAX_OBJECTS], istack[MAX_IP_OBJECTS], pstack[MAX_OBJECTS];
54 Aobject astack[MAX_OBJECTS];
55 char input_list[STRING_DIM];
56 int k, closed_squares, original_depth;
57
```

```

58 void initializeObject(Object stack[], int dim) {
59     for(int j=0; j < dim; j++) {
60         stack[j].leng = -1;
61     }
62 }
63
64 void initializeStringArray(char array[]) {
65     for(int j=0; j < STRING_DIM; j++) {
66         array[j] = '\0';
67     }
68 }
69
70 void error(char *msg) {
71     printf("%s", msg);
72     exit(-1);
73 }
74
75 void checkip() {
76     while(istack[ip].leng != -1) {
77         ip++;
78     }
79 }
80
81 void push_obj(Object obj) {
82     if(op >= MAX_OBJECTS)
83         error("Object stack overflow \n");
84     ostack[op++] = obj;
85 }
86
87 Object pop_obj() {
88     return ostack[--op];
89 }
90
91
92 void push_int(int n) {
93     Object obj;
94
95     obj.type = D_INT;
96     obj.leng = 0;
97     obj.inst.ival = n;
98     push_obj(obj);
99 }
100
101 void push_string(char *str) {
102     Object obj;
103
104     obj.type = D_STRING;
105     obj.leng = 0;
106     obj.inst.sval = malloc(strlen(str));
107     strcpy(obj.inst.sval,str);
108
109     push_obj(obj);
110 }
111
112 void push_bool(int b) {
113     Object obj;
114
115     obj.type = D_BOOL;
116     obj.leng = 0;
117     obj.inst.ival = b;
118     push_obj(obj);
119 }
120
121 void push_list(int leng) {
122     Object obj;
123
124     obj.type = D_LIST;
125     obj.leng = leng;

```

```

126     if(leng == 0)
127     |
128     |   obj.inst.ival = -1;
129     |
130     |   obj.inst.ival = ip;
131     |
132     |   op = op - obj.leng;
133     |
134     |   for(int j=0; j < obj.leng; j++) {
135     |   |   istack[ip++] = ostack[op++];
136     |   }
137     |   op = op - obj.leng;
138     |
139     |   push_obj(obj);
140   }
141   Object pop_list() {
142   |   Object obj;
143   |
144   |   obj = pop_obj();
145   |   ip = obj.inst.ival;
146   |   return obj;
147   }
148
149   void exec_loci(int n) {
150   |   push_int(n);
151   }
152
153   void exec_locb(int b) {
154   |   push_bool(b);
155   }
156
157   void exec_locs(char *s) {
158   |   push_string(s);
159   }
160
161   void exec_list(int leng) {
162   |   push_list(leng);
163   }
164
165
166   void exec_vars(int num_vars) {
167   |   vars = malloc(num_vars*sizeof(Object));
168   |   size = num_vars;
169   }
170
171   void exec_newi() {
172   |   vars[vp].type = D_INT;
173   |   vars[vp].leng = 0;
174   |   vars[vp].inst.ival = -1;
175   |   vp++;
176   }
177
178   void exec_newb() {
179   |   vars[vp].type = D_BOOL;
180   |   vars[vp].leng = 0;
181   |   vars[vp].inst.ival = -1;
182   |   vp++;
183   }
184
185   void exec_news() {
186   |   vars[vp].type = D_STRING;
187   |   vars[vp].leng = 0;
188   |   vars[vp].inst.sval = "null";
189   |   vp++;
190   }
191
192   void exec_newl() {
193   |   vars[vp].type = D_LIST;

```

```

194     |     vars[vp].leng = 0;
195     |     vars[vp].inst.ival = -1;
196     |     vp++;
197   }
198
199 void exec_load(int flag, int oid) {
200   if(flag == 0) {
201     |     push_obj(vars[oid]);
202   }
203   else {
204     |     int current_param = pp - astack[ap].num;
205     |     push_obj(pstack[current_param + oid]);
206   }
207 }
208
209 void exec_stor(int oid) {
210   |     vars[oid] = pop_obj();
211 }
212
213 void exec_skip(int offset) {
214   |     pc += offset-1;
215 }
216
217 void exec_skpf(int offset) {
218   |     if(pop_obj().inst.ival == FALSE)
219   |       |     pc += offset-1;
220 }
221
222 int equal(Object obj1, Object obj2) {
223   switch(obj1.type) {
224     case D_BOOL:
225     case D_INT:
226       |       return (obj1.inst.ival == obj2.inst.ival);
227     case D_STRING:
228       |       return (strcmp(obj1.inst.sval, obj2.inst.sval) == 0);
229     case D_LIST:
230       |       if(obj1.leng != obj2.leng)
231       |         |       return FALSE;
232       |       for(int i=0; i < obj1.leng; i++) {
233         |         if(!equal(istack[obj1.inst.ival + i], istack[obj2.inst.ival + i])) {
234         |           |       return FALSE;
235         }
236       }
237       |       return TRUE;
238   }
239 }
240
241 void exec_equa() {
242   Object obj1, obj2;
243
244   obj2 = pop_obj();
245   obj1 = pop_obj();
246   push_bool(equal(obj1, obj2));
247   if(obj1.type == D_LIST)
248     |     ip = obj2.inst.ival + obj2.leng;
249 }
250
251 void exec_nequ() {
252   Object obj1, obj2;
253
254   obj2 = pop_obj();
255   obj1 = pop_obj();
256   push_bool(!equal(obj1, obj2));
257   if(obj1.type == D_LIST)
258     |     ip = obj2.inst.ival + obj2.leng;
259 }
260
261 void exec_grtr() {

```

```

262     Object obj1, obj2;
263
264     obj2 = pop_obj();
265     obj1 = pop_obj();
266     if(obj1.type == D_INT)
267     |   push_bool(obj1.inst.ival > obj2.inst.ival);
268     else
269     |   push_bool(strcmp(obj1.inst.sval, obj2.inst.sval) > 0);
270
271 }
272
273 void exec_greq() {
274     Object obj1, obj2;
275
276     obj2 = pop_obj();
277     obj1 = pop_obj();
278     if(obj1.type == D_INT)
279     |   push_bool(obj1.inst.ival >= obj2.inst.ival);
280     else
281     |   push_bool(strcmp(obj1.inst.sval, obj2.inst.sval) >= 0);
282
283 }
284
285 void exec_leth() {
286     Object obj1, obj2;
287
288     obj2 = pop_obj();
289     obj1 = pop_obj();
290     if(obj1.type == D_INT)
291     |   push_bool(obj1.inst.ival < obj2.inst.ival);
292     else
293     |   push_bool(strcmp(obj1.inst.sval, obj2.inst.sval) < 0);
294
295 }
296
297 void exec_leq() {
298     Object obj1, obj2;
299
300     obj2 = pop_obj();
301     obj1 = pop_obj();
302     if(obj1.type == D_INT)
303     |   push_bool(obj1.inst.ival <= obj2.inst.ival);
304     else
305     |   push_bool(strcmp(obj1.inst.sval, obj2.inst.sval) <= 0);
306
307 }
308
309 void exec_plus() {
310     int n,m;
311     n = pop_obj().inst.ival;
312     m = pop_obj().inst.ival;
313     push_int(m+n);
314 }
315
316 void exec_minu() {
317     int n,m;
318     n = pop_obj().inst.ival;
319     m = pop_obj().inst.ival;
320     push_int(m-n);
321 }
322
323 void exec_mult() {
324     int n,m;
325     n = pop_obj().inst.ival;
326     m = pop_obj().inst.ival;
327     push_int(m*n);
328 }
329

```

```

330 void exec_divi() {
331     int n,m;
332     n = pop_obj().inst.ival;
333     m = pop_obj().inst.ival;
334     push_int(m/n);
335 }
336
337 void exec_app() {
338     Object obj1, obj2, obj;
339     obj2 = pop_list();
340     obj1 = pop_list();
341     obj.type = D_LIST;
342     obj.leng = obj1.leng + obj2.leng;
343
344     if(obj1.leng == 0) {
345         ip = obj2.inst.ival + obj2.leng;
346         checkip();
347         push_obj(obj2);
348         return;
349     }
350     if(obj2.leng == 0) {
351         ip = obj1.inst.ival + obj1.leng;
352         checkip();
353         push_obj(obj1);
354         return;
355     }
356
357     checkip();
358     int abs_ip = ip;
359     obj.inst.ival = abs_ip;
360
361     ip = obj1.inst.ival;
362     for(int i=0; i < obj1.leng; i++) {
363         istack[abs_ip++] = istack[ip+i];
364     }
365
366     ip = obj2.inst.ival;
367     for(int i=0; i < obj2.leng; i++) {
368         istack[abs_ip++] = istack[ip+i];
369     }
370
371     ip = abs_ip;
372     push_obj(obj);
373 }
374
375 void exec_umin() {
376     int n;
377     n = pop_obj().inst.ival;
378     push_int(-n);
379 }
380
381 void exec_rang() {
382     int start, end, leng;
383     end = pop_obj().inst.ival;
384     start = pop_obj().inst.ival;
385     if(start > end) {
386         leng = 0;
387     }
388     else_if (start == end) {
389         leng = 1;
390         push_int(start);
391     }
392     else {
393         leng = end - start + 1;
394         while(start <= end) {
395             push_int(start++);
396         }
397     }

```

```

398     |     push_list(leng);
399 }
400
401 void exec_nega() {
402     int b;
403     b = pop_obj().inst.ival;
404     if(b == TRUE)
405         b = FALSE;
406     else
407         b = TRUE;
408     push_bool(b);
409 }
410
411 void exec_empt() {
412     Object obj;
413     int abs_ip = ip;
414     obj = pop_list();
415     push_bool(obj.leng==0);
416     ip = abs_ip;
417 }
418
419 void exec_leng() {
420     Object obj;
421     int abs_ip = ip;
422     obj = pop_list();
423     push_int(obj.leng);
424     ip = abs_ip;
425 }
426
427 void exec_head() {
428     Object obj, h;
429     obj = pop_list();
430     if(obj.leng==0)
431         error("Function head applied to empty list \n");
432
433     h = istack[obj.inst.ival];
434     push_obj(h);
435
436     ip = obj.leng + obj.inst.ival;
437     checkip();
438 }
439
440 void exec_tail() {
441     Object obj, t;
442     obj = pop_list();
443     if(obj.leng==0)
444         error("Function tail applied to empty list \n");
445     t = obj;
446     t.inst.ival = obj.inst.ival + 1;
447     t.leng = obj.leng - 1;
448     push_obj(t);
449
450     ip = obj.leng + obj.inst.ival;
451     checkip();
452 }
453
454 void exec_last() {
455     Object obj, last;
456     obj = pop_list();
457     if(obj.leng==0)
458         error("Function last applied to empty list \n");
459
460     last = istack[obj.leng + obj.inst.ival - 1];
461     push_obj(last);
462
463     ip = obj.leng + obj.inst.ival;
464     checkip();
465 }

```

```

466
467 void exec_init() {
468     Object obj, init;
469     obj = pop_list();
470     if(obj.leng==0)
471         error("Function init applied to empty list \n");
472     init = obj;
473     init.leng = obj.leng - 1;
474     push_obj(init);
475
476     ip = obj.leng + obj.inst.ival;
477     checkip();
478 }
479
480
481 void exec_push(int params) {
482     astack[+ap].num = params;
483     astack[ap].objs = op - astack[ap].num;
484     for(int i=0; i < params; i++)
485         pstack[pp++] = ostack[astack[ap].objs + i];
486
487     op = astack[ap].objs;
488 }
489
490 void exec_jump(int address) {
491     astack[ap].ret = pc;
492     pc = address;
493 }
494
495 void exec_retn() {
496     pc = astack[ap].ret;
497 }
498
499 void exec_apop() {
500     pp = pp - astack[ap--].num;
501 }
502
503 void exec_geti() {
504     int input;
505     scanf("%d", &input);
506     push_int(input);
507 }
508
509 void exec_gets() {
510     char input[STRING_DIM];
511     fgets(input, STRING_DIM, stdin);
512     push_string(input);
513 }
514
515 void exec_getb() {
516     int b;
517     char input[STRING_DIM];
518     scanf("%s", input);
519     if(strcmp("true", input) == 0)
520         b = 1;
521     else if(strcmp("false", input) == 0)
522         b = 0;
523     else
524         error("Attention: wrong boolean format\n");
525     push_bool(b);
526 }
527
528 void push_atomic(int atomic, char s[]) {
529     switch(atomic) {
530         case A_INT: {
531             int n = atoi(s);
532             if(n == 0 && strcmp(s, "0")!=0) {
533                 fprintf(stderr, "Attention: \"%s\" is not an

```

```

534     |     |     |     |     int element in the list inserted\n", s);
535     |     |     |     |     exit(-1);
536     |     |     |
537     |     |     else
538     |     |     |     push_int(n);
539     |     |     break;
540     |
541     |     case A_STRING: {
542     |     |     push_string(s);
543     |     |     break;
544     |     }
545     |     case A_BOOL: {
546     |     |     int b;
547     |     |     if(strcmp("true", s) == 0)
548     |     |     |     b = 1;
549     |     |     else_if(strcmp("false", s) == 0)
550     |     |     |     b = 0;
551     |     |     else {
552     |     |     |     fprintf(stderr, "Attention: \"%s\" is not a boolean
553     |     |     |     element in the list inserted\n", s);
554     |     |     |     exit(-1);
555     |     |     }
556     |     |     push_bool(b);
557     |     |     break;
558     |     }
559     |
560   }
561
562 void read_empty_list() {
563   k++;
564   int v=k;
565   push_list(0);
566   closed_squares = original_depth-1;
567   while(input_list[v] == '[' && input_list[v+1] != '\0') {
568     closed_squares--;
569     v++;
570   }
571 }
572
573 void read_list_with_depth_1(int atomic, char input_list[]) {
574   int leng = 0;
575   int i = 0;
576   char elem[STRING_DIM];
577   initializeStringArray(elem);
578   while(input_list[k] != ']' && input_list[k] != '\0') {
579     if(input_list[k] == '[') {
580       error("Attention: there is an extra '[' that is not required \n");
581     }
582     if(input_list[k] != ',') {
583       elem[i++] = input_list[k];
584     }
585     else {
586       if(input_list[k] == ',' && input_list[k+1] == ']') {
587         error("Attention: after ',' there must not be a ']' \n");
588       }
589       push_atomic(atomic, elem);
590       i = 0;
591       leng++;
592       for(int j=0; j < STRING_DIM; j++)
593         |     elem[j] = '\0';
594     }
595     k++;
596   }
597   if(input_list[k] != ']') {
598     error("Attention: the last character of a list must be ']' \n");
599   }
600   push_atomic(atomic, elem);
601   leng++;

```

```

602     k++;
603     closed_squares++;
604     push_list(leng);
605 }
606
607 void check_last_list_char(int index) {
608     if(input_list[index] == '\0') {
609         error("Attention: the last character of a list must be ']' \n");
610     }
611 }
612
613 void read_list_with_depth_greater_then_1(int depth, int atomic) {
614     int leng = 1;
615     if(input_list[k++] != '[') {
616         check_last_list_char(k-1);
617         printf("Attention: character expected is '[' and not %c", input_list[k-1]);
618         error("\n");
619     }
620     if(input_list[k-1] == '[' && input_list[k] == ']') {
621         read_empty_list();
622         return;
623     }
624     if(depth == 1) {
625         read_list_with_depth_1(atomic, input_list);
626         return;
627     }
628     while(closed_squares < depth) {
629         read_list_with_depth_greater_then_1(depth-1, atomic);
630         if(input_list[k] == ',') {
631             leng++;
632             closed_squares = 0;
633         }
634         else if(input_list[k] == ']') {
635             push_list(leng);
636             closed_squares++;
637         }
638         else {
639             check_last_list_char(k);
640             printf("Attention: character expected is ']' or ',' and not %c", input_list[k]);
641             error("\n");
642         }
643         k++;
644     }
645 }
646
647 void exec_getl(int depth, int atomic) {
648     initializeStringArray(input_list);
649     scanf("%s", input_list);
650     original_depth = depth;
651     read_list_with_depth_greater_then_1(depth, atomic);
652     if(input_list[k] != '\0') {
653         error("Attention: there must be no characters after the last ']' \n");
654     }
655     closed_squares = 0;
656     k=0;
657 }
658
659 void exec_inst_puts_show() {
660     Object instance = istack[ip++];
661
662     if(instance.type == D_LIST) {
663         int next_ip = ip;
664         ip = instance.inst.ival;
665         printf("[");
666         for(int j=0; j < instance.leng; j++) {
667             exec_inst_puts_show();
668             if(j != instance.leng-1)
669                 printf(",");

```

```

670     }
671     printf("]");
672     ip = next_ip;
673 }
674 else {
675     if(instance.type == D_STRING)
676         printf("%s", instance.inst.sval);
677     else if(instance.type == D_INT)
678         printf("%d", instance.inst.ival);
679     else
680         printf("%s", instance.inst.ival == 1 ? "true" : "false");
681 }
682 }
683
684 void exec_puts_show() {
685     int abs_ip = ip;
686     Object obj = pop_obj();
687
688     if(obj.type == D_LIST) {
689         op++;
690         obj = pop_list();
691         printf("[");
692         for(int j=0; j < obj.leng; j++) {
693             exec_inst_puts_show();
694             if(j != obj.leng-1)
695                 printf(",");
696         }
697         printf("]");
698         ip = abs_ip;
699     }
700     else {
701         if(obj.type == D_STRING) {
702             printf("%s", obj.inst.sval);
703         }
704         else if(obj.type == D_INT)
705             printf("%d", obj.inst.ival);
706         else
707             printf("%s", obj.inst.ival == 1 ? "true" : "false");
708     }
709
710     if(&prog[pc-1])->oper == PPUTS
711     op++;
712 }
713
714 void execute(Tcode *tstat) {
715     switch(tstat->oper) {
716         case VARS:
717             exec_vars(tstat->args[0].ival);
718             break;
719         case NEWI:
720             exec_newi();
721             break;
722         case NEWS:
723             exec_news();
724             break;
725         case NEWB:
726             exec_newb();
727             break;
728         case NEWL:
729             exec_newl();
730             break;
731         case LOCI:
732             exec_loci(tstat->args[0].ival);
733             break;
734         case LOCB:
735             exec_locb(tstat->args[0].ival);
736             break;
737         case LOCS:

```

```

738     exec_locs(tstat->args[0].sval);
739     break;
740 case LIST:
741     exec_list(tstat->args[0].ival);
742     break;
743 case LOAD:
744     exec_load(tstat->args[0].ival, tstat->args[1].ival);
745     break;
746 case STOR:
747     exec_stor(tstat->args[0].ival);
748     break;
749 case SKPF:
750     exec_skpf(tstat->args[0].ival);
751     break;
752 case SKIP:
753     exec_skip(tstat->args[0].ival);
754     break;
755 case EQUA:
756     exec_equa();
757     break;
758 case NEQU:
759     exec_nequ();
760     break;
761 case GRTR:
762     exec_grtr();
763     break;
764 case GREQ:
765     exec_greq();
766     break;
767 case LETH:
768     exec_leth();
769     break;
770 case LEEQ:
771     exec_leeq();
772     break;
773 case PLUS:
774     exec_plus();
775     break;
776 case MINU:
777     exec_minu();
778     break;
779 case MULT:
780     exec_mult();
781     break;
782 case DIVI:
783     exec_divi();
784     break;
785 case APPN:
786     exec_app();
787     break;
788 case UMIN:
789     exec_umin();
790     break;
791 case NEGA:
792     exec_nega();
793     break;
794 case RANG:
795     exec_rang();
796     break;
797 case T_EMPT:
798     exec_empt();
799     break;
800 case T LENG:
801     exec_leng();
802     break;
803 case T HEAD:
804     exec_head();
805     break;

```

```

806     case T_TAIL:
807         exec_tail();
808         break;
809     case T_LAST:
810         exec_last();
811         break;
812     case T_INIT:
813         exec_init();
814         break;
815     case PUSH:
816         exec_push(tstat->args[0].ival);
817         break;
818     case JUMP:
819         exec_jump(tstat->args[0].ival);
820         break;
821     case APOP:
822         exec_apop();
823         break;
824     case T_FUNC: break;
825     case RETN:
826         exec_retn();
827         break;
828     case GETI:
829         exec_geti();
830         break;
831     case GETS:
832         exec_gets();
833         break;
834     case GETB:
835         exec_getb();
836         break;
837     case GETL:
838         exec_getl(tstat->args[0].ival, tstat->args[1].ival);
839         break;
840     case PUTS:
841         exec_puts_show();
842         printf("\n");
843         break;
844     case T_SHOW:
845         exec_puts_show();
846         break;
847     default: error("Unknown T-code instruction \n");
848         break;
849     }
850 }
851
852 Operator fromStringToOp(char *str) {
853     if(strcmp(str,"VARS")==0)
854         return VARS;
855     else if(strcmp(str,"HALT")==0)
856         return HALT;
857     else if(strcmp(str,"NEWI")==0)
858         return NEWI;
859     else if(strcmp(str,"NEWB")==0)
860         return NEWB;
861     else if(strcmp(str,"NEWS")==0)
862         return NEWS;
863     else if(strcmp(str,"NEWL")==0)
864         return NEWL;
865     else if(strcmp(str,"LOCI")==0)
866         return LOCI;
867     else if(strcmp(str,"LOCS")==0)
868         return LOCS;
869     else if(strcmp(str,"LOCB")==0)
870         return LOCB;
871     else if(strcmp(str,"LIST")==0)
872         return LIST;
873     else if(strcmp(str,"LOAD")==0)

```

```

874     |     return LOAD;
875     |     else if(strcmp(str,"STOR")==0)
876     |         return STOR;
877     |     else if(strcmp(str,"SKPF")==0)
878     |         return SKPF;
879     |     else if(strcmp(str,"SKIP")==0)
880     |         return SKIP;
881     |     else if(strcmp(str,"EQUA")==0)
882     |         return EQUA;
883     |     else if(strcmp(str,"NEQU")==0)
884     |         return NEQU;
885     |     else if(strcmp(str,"GRTR")==0)
886     |         return GRTR;
887     |     else if(strcmp(str,"GREQ")==0)
888     |         return GREQ;
889     |     else if(strcmp(str,"LETH")==0)
890     |         return LETH;
891     |     else if(strcmp(str,"LEEQ")==0)
892     |         return LEEQ;
893     |     else if(strcmp(str,"PLUS")==0)
894     |         return PLUS;
895     |     else if(strcmp(str,"MINU")==0)
896     |         return MINU;
897     |     else if(strcmp(str,"MULT")==0)
898     |         return MULT;
899     |     else if(strcmp(str,"DIVI")==0)
900     |         return DIVI;
901     |     else if(strcmp(str,"APPN")==0)
902     |         return APPN;
903     |     else if(strcmp(str,"UMIN")==0)
904     |         return UMIN;
905     |     else if(strcmp(str,"NEGA")==0)
906     |         return NEGA;
907     |     else if(strcmp(str,"RANG")==0)
908     |         return RANG;
909     |     else if(strcmp(str,"EMPT")==0)
910     |         return T_EMPT;
911     |     else if(strcmp(str,"LENG")==0)
912     |         return T LENG;
913     |     else if(strcmp(str,"HEAD")==0)
914     |         return T_HEAD;
915     |     else if(strcmp(str,"TAIL")==0)
916     |         return T_TAIL;
917     |     else if(strcmp(str,"LAST")==0)
918     |         return T_LAST;
919     |     else if(strcmp(str,"INIT")==0)
920     |         return T_INIT;
921     |     else if(strcmp(str,"PUSH")==0)
922     |         return PUSH;
923     |     else if(strcmp(str,"JUMP")==0)
924     |         return JUMP;
925     |     else if(strcmp(str,"FUNC")==0)
926     |         return T_FUNC;
927     |     else if(strcmp(str,"APOP")==0)
928     |         return APOP;
929     |     else if(strcmp(str,"RETN")==0)
930     |         return RETN;
931     |     else if(strcmp(str,"GETI")==0)
932     |         return GETI;
933     |     else if(strcmp(str,"GETS")==0)
934     |         return GETS;
935     |     else if(strcmp(str,"GETB")==0)
936     |         return GETB;
937     |     else if(strcmp(str,"GETL")==0)
938     |         return GETL;
939     |     else if(strcmp(str,"PUTS")==0)
940     |         return PUTS;
941     |     else if(strcmp(str,"SHOW")==0)

```

```

942     |     return T_SHOW;
943   |   else
944   |     error("Unknown T-code instruction\n");
945 }
946
947 Atomic fromStringToAtomic(char *arg) {
948   if(strcmp(arg, "INT")==0)
949     return A_INT;
950   else if(strcmp(arg, "STRING")==0)
951     return A_STRING;
952   else if(strcmp(arg, "BOOL")==0)
953     return A_BOOL;
954
955   return A_UNKNOWN;
956 }
957
958 void initialize_char_array(char array[]) {
959   for(int i=0; i < STRING_DIM; i++) {
960     array[i] = '\0';
961   }
962 }
963
964 void start_machine() {
965   FILE *fp;
966   fp = fopen("IntermediateCode.txt", "r");
967
968   char word[STRING_DIM];
969   int i = 0, step = 0, code_size;
970   char c = fgetc(fp);
971
972   while(c != '\n') {
973     while(c != ' ' && c != '\n') {
974       word[i++] = c;
975       c=fgetc(fp);
976     }
977     if(step != 0)
978       code_size = atoi(word);
979     else
980       c=fgetc(fp);
981     step++;
982     initialize_char_array(word);
983     i = 0;
984   }
985   step = 0;
986   i = 0;
987   initialize_char_array(word);
988   prog = (Tcode*) malloc(code_size*sizeof(Tcode));
989
990   while((c = fgetc(fp)) != EOF) {
991     while(c != '\n') {
992       while(c != ' ' && c != '\n') {
993         if(c == '\"') {
994           c=fgetc(fp);
995           while(c != '\"') {
996             word[i++] = c;
997             c=fgetc(fp);
998             if(word[i-2] == '\\\' && word[i-1] == '\n') {
999               i = i-2;
1000               word[i++] = '\n';
1001               word[i] = '\0';
1002             }
1003           }
1004           c=fgetc(fp);
1005         }
1006         else {
1007           word[i++] = c;
1008           c=fgetc(fp);
1009         }

```

```

1010     }
1011     if(step == 0)
1012     |     prog->oper = fromStringToOp(word);
1013     else if(step == 1) {
1014         if(prog->oper == LOCS) {
1015             prog->args[0].sval = malloc(strlen(word));
1016             strcpy(prog->args[0].sval,word);
1017         }
1018         else
1019             prog->args[0].ival = atoi(word);
1020     }
1021     else {
1022         if(prog->oper == GETL) {
1023             prog->args[1].ival = fromStringToAtomic(word);
1024         }
1025         else
1026             prog->args[1].ival = atoi(word);
1027     }
1028
1029     step++;
1030     i = 0;
1031     initialize_char_array(word);
1032     if(c != '\n')
1033         c=fgetc(fp);
1034 }
1035     step = 0;
1036     i = 0;
1037     initialize_char_array(word);
1038     prog++;
1039 }
1040
1041     prog = prog - code_size;
1042     fclose(fp);
1043
1044     initializeObject(istack, MAX_IP_OBJECTS);
1045
1046 }
1047
1048 int main() {
1049     Tcode *tstat;
1050
1051     start_machine();
1052     printf("\n");
1053     while((tstat = &prog[pc++])->oper != HALT) {
1054         execute(tstat);
1055     }
1056     printf("\n\n");
1057     return 0;
1058 }
1059

```

Ringraziamenti

Vorrei dedicare qualche parola di ringraziamento a tutti coloro che mi hanno sostenuto durante questi anni di studio. Innanzitutto, i miei genitori che, più di qualsiasi insegnante, mi hanno fatto capire l'importanza dell'istruzione e mi hanno incoraggiato a seguire le mie inclinazioni, senza mai impormi nulla. Sono sempre stati presenti per me, sia nelle occasioni banali che in quelle importanti, e ho la certezza che continueranno ad esserlo anche in futuro; se non fosse stato per loro, sicuramente non sarei la persona che sono oggi e non avrei raggiunto questo traguardo. Mio papà mi ha insegnato che il risultato non è tutto, ma conta come lo si ottiene: lealmente e col duro lavoro. Da mia mamma ho appreso importanti lezioni di vita, certamente applicabili in ambito scolastico, ma che hanno una valenza più universale e profonda. I miei nonni, che hanno sempre creduto in me e che aspettavano l'esito di un esame con la mia stessa impazienza, gioendo quasi più di me quando prendevo un bel voto. La mia fidanzata, con la quale ho condiviso le gioie e le fatiche del percorso universitario. Sicuramente l'esperienza universitaria non sarebbe stata altrettanto memorabile se non ti avessi conosciuta e sono emozionato all'idea di vivere questo traguardo insieme a te. Il mio fratellino e il resto della mia famiglia, anche i membri non più presenti tra noi purtroppo, che si sono sempre interessati ai miei studi e che mi hanno continuamente incoraggiato. Infine un grazie ai miei amici più stretti, coi quali mi sono goduto appieno questi anni universitari, in ogni loro sfaccettatura.

Con immensa soddisfazione,
Dario