Compilatori e Interpreti

Relazione Progetto
SIMPLAN PLUS

Selene Gerali
Federico Pennino
Cristiana Angiuoni

21 Giugno 2021

Indice

1 Introduzione

Il progetto è stato sviluppato con lo scopo di:

1. realizzare un sistema di analisi semantica per il linguaggio `SimpLan Plus`, tenendo in considerazione anche l'analisi degli effetti.
2. definire un linguaggio bytecode per eseguire programmi scritti in `SimpLan Plus`.
3. generare il codice e implementare l'interprete per il bytecode

Il linguaggio di programmazione che è stato utilizzato per sviluppare l'intero progetto è Java 8, con il supporto della libreria ANTLR 4, strumento generatore di lexer e parser, necessario per analizzare sintatticamente e semanticamente testi strutturati.
Ogni classe che implementa l'interfaccia `Node`, corrisponde a un non terminale della grammatica. La struttura generale di tali classi è rappresentata dallo schema in Figura 1.

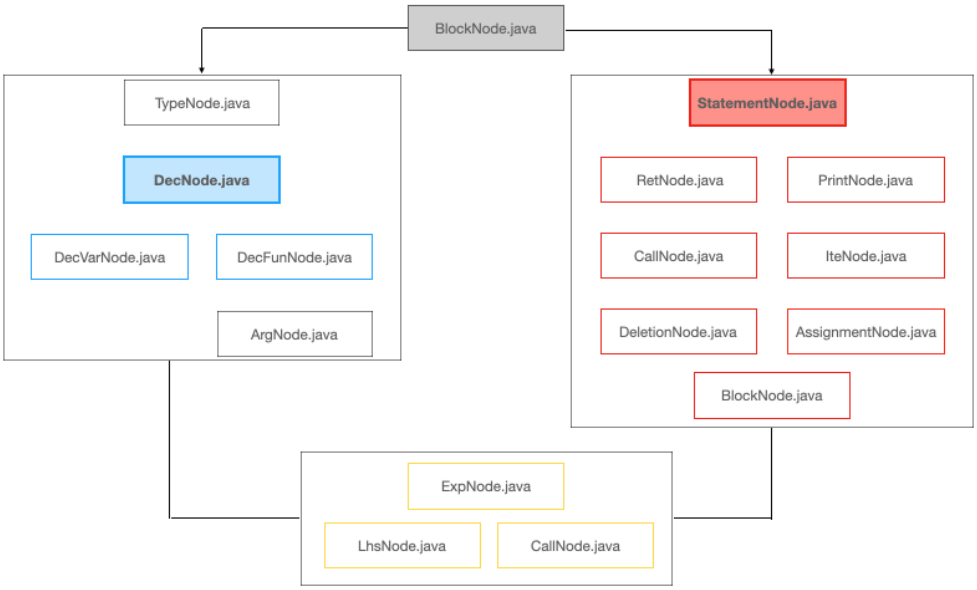

Figura 1: Schema delle classi

Ogni programma scritto secondo le regole della grammatica di `SimpLan Plus` inizia con un blocco (sintatticamente delimitato da parentesi graffe), all'interno del quale possono essere presenti dichiarazioni e comandi. La sequenza di dichiarazioni può essere espressa alternando in ordine arbitrario sia dichiarazioni di variabili che di funzioni, ma può essere anche vuota. Il vincolo necessario affinché il programma soddisfi la regola definita dalla grammatica per il non terminale *block* è che, in ogni caso, le dichiarazioni devono precedere la sequenza dei comandi, se presente. Poiché uno dei comandi accettati dalla grammatica è block stesso, il programma può essere strutturato in blocchi annidati.

Nei capitoli successivi vengono presentate le fasi principali attraversate durante lo sviluppo del progetto.
Nella sezione 2 vengono descritte le principali scelte implementative, con attenzione particolare alle strutture dati necessarie per la rappresentazione della tabella dei simboli e della memoria. Il modo in cui è stata effettuata l'analisi semantica dei tipi viene spiegato nella sezione 3, attraverso la definizione delle regole di inferenza. Nella sezione 4 viene descritta nel dettaglio la fase di analisi degli effetti relativa al corretto utilizzo di particolari istruzioni. La sezione 5 presenta la fase di generazione del codice e, infine, nella sezione 6 si conclude con la fase di testing, attraverso la creazione di una batteria di test, in cui vengono mostrati alcuni esempi di esecuzione sia di programmi corretti che errati.

2 Scelte implementative

2.1 Struttura della tabella dei simboli

È stato scelto di implementare la tabella dei simboli relativa a ogni blocco attraverso un'istanza della classe `SymTable`. Ogni oggetto di tale classe contiene una HashMap in cui vengono memorizzati i tipi relativi agli identificatori presenti nello scope corrente (nomi di variabili, nomi di costanti, nomi di funzioni). In questo modo è possibile effettuare l'accesso alla tabella in modo efficiente attraverso il nome dell'identificatore. Oltre a questa struttura dati, un oggetto di tipo `SymTable` contiene il riferimento alla tabella dei simboli relativa alle dichiarazioni dello scope immediatamente precedente.

La tabella dei simboli del programma, (Figura 2), è utilizzata per verificarne la correttezza semantica. In particolare, per ogni dichiarazione, viene effettuata una ricerca nella HashMap relativa allo scope corrente per evitare dichiarazioni multiple nello stesso livello di annidamento. Inoltre, dato l'uso di un identificatore, viene utilizzata la tabella dei simboli per verificare l'esistenza della sua dichiarazione nello scope corrente o in quelli precedenti. Per implementare questo meccanismo, è stata utilizzata la regola dell'annidamento più vicino. Nel caso in cui la dichiarazione del nome non venga trovata, la ricerca all'interno della tabella dei simboli fallisce e viene sollevata un'eccezione.

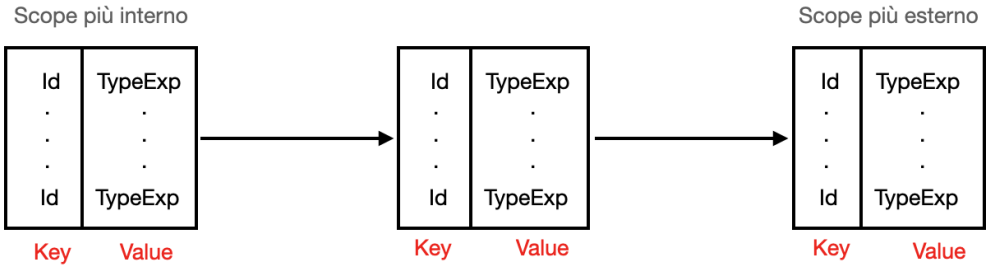

Figura 2: Struttura della SymTable di un programma con tre blocchi annidati

Per semplicità e maggior chiarezza, è stato scelto di creare una nuova tabella dei simboli utilizzata in fase di generazione del codice, chiamata `OffsetSymTable` (Figura 3). Questa tabella memorizza l'offset di ogni singolo identificatore. L'offset è un numero intero che indica la distanza tra due identificatori all'interno della pila. Ogni identificatore occupa 1 byte, quindi la somma degli offset corrisponde alla cardinalità dell'intera tabella dei simboli del programma.
Inoltre, nel caso in cui l'identificatore corrisponda al nome di una funzione, è stato scelto di memorizzare l'etichetta relativa ad essa, per semplificare la ricerca delle definizioni di funzioni in fase di generazione del codice.

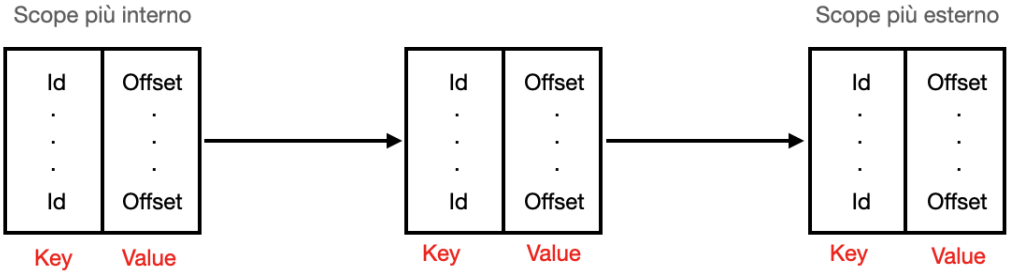

Figura 3: Struttura della OffsetSymTable di un programma con tre blocchi annidati

La tabella dei simboli relativa all'analisi degli effetti, chiamata `EffectSymTable` (Figura4), è utilizzata per memorizzare il cambiamento di stato di ciascun identificatore.
Il dominio dei tipi effetto è:

$$B = \{Bottom, Rw, Delete, Top\}$$

B è un ordine parziale con la seguente relazione d'ordine:

$$Bottom \leq Rw \leq Delete \leq Top$$

Gli effetti di un identificatore possono assumere i seguenti tipi:

- **Bottom**: effetto memorizzato in fase di dichiarazione di un identificatore, prima di qualunque suo utilizzo.
- **Rw**: effetto memorizzato in fase di utilizzo di un identificatore (accesso in lettura o scrittura).
- **Delete**: effetto memorizzato in fase di cancellazione di un identificatore.
- **Top:** effetto memorizzato in fase di violazione della relazione d'ordine dei tipi effetto.

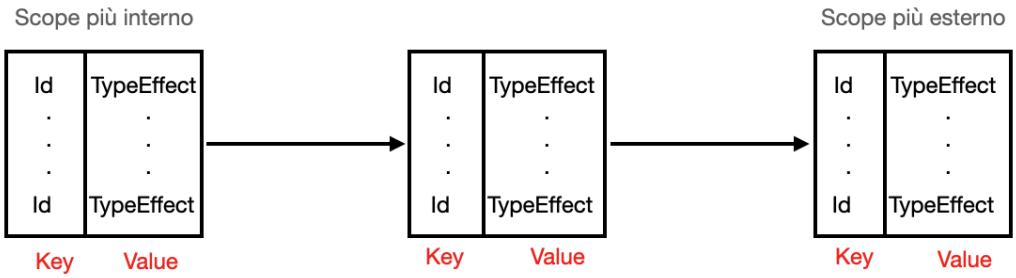

Figura 4: Struttura della EffectSymTable di un programma con tre blocchi annidati

2.2 Struttura della memoria

Nella fase di generazione del codice viene utilizzata la memoria, costituita da stack e heap (Figura 5). Lo stack viene rappresentato attraverso l'allocazione di record di attivazione, ognuno dei quali rappresenta un blocco del programma e deve contenere le informazioni necessarie affinchè l'intera esecuzione termini correttamente.

In questo progetto si è scelto di strutturare ogni record di attivazione attraverso la memorizzazione sulla pila delle seguenti informazioni:

- Frame pointer
- Variabili locali statiche
- Return address

Inoltre, ogni cella di memoria, indipendentemente dal valore memorizzato al suo interno, è stata considerata della dimensione di 1 byte. Per questo motivo, la dimensione totale della memoria coincide con il massimo valore assegnabile a un indirizzo, il quale corrisponde all'ultimo elemento dello stack. Poichè le attivazioni dei blocchi sono nidificate, lo stack tiene traccia di tutti i record attivi e, alla chiusura di un blocco, il relativo frame viene completamente deallocato.

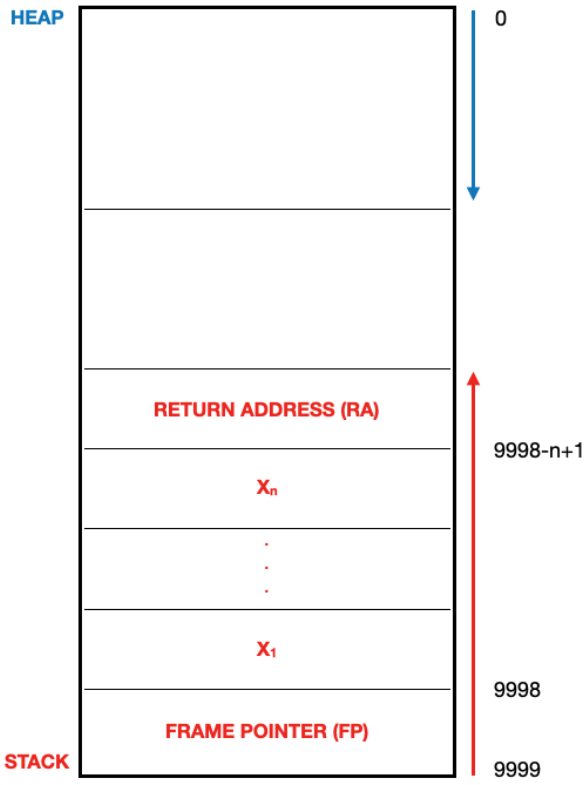

Figura 5: Struttura generale della memoria

Per memorizzare le variabili di tipo puntatore, è necessario l'utilizzo dell'heap. Questo meccanismo permette di evitare che il valore della variabile allocata dinamicamente venga perso alla chiusura del blocco (Figura 6).

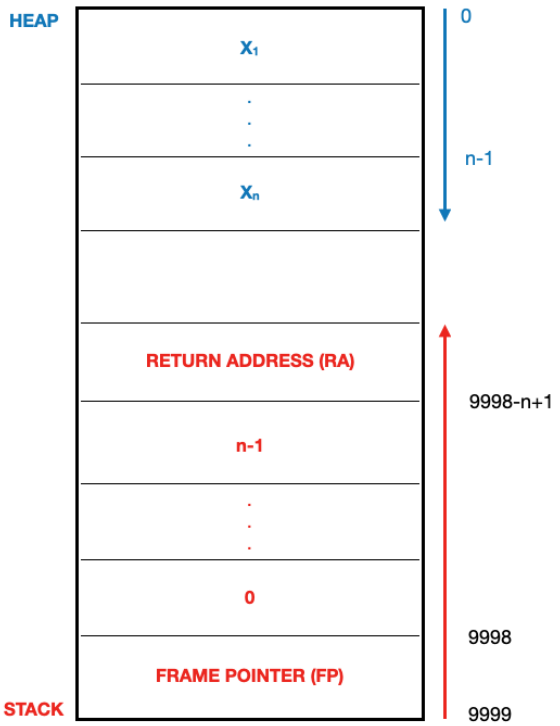

Figura 6: Struttura della memoria nel caso di puntatori

La struttura dello stack dopo una chiamata di funzione è mostrata in (Figura 7). A ogni chimata di funzione presente nel programma, viene creato un nuovo record di attivazione contenente, oltre al frame pointer e al return address, anche i parametri attuali. Inoltre, viene creato un ulteriore frame che assume la struttura tradizionale precedentemente descritta, dedicato al corpo della funzione. In tale frame, se la funzione ha un valore di ritorno, questo viene memorizzato all'indirizzo immediatamente precedente rispetto al return address.

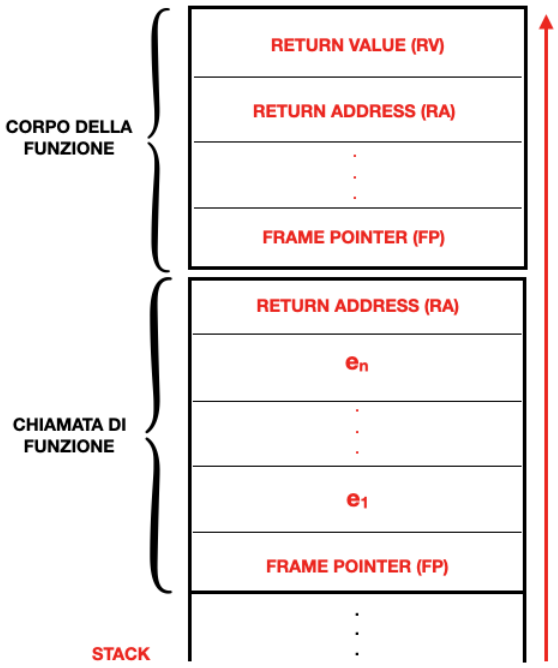

Figura 7: Struttura della memoria nel caso di chiamata di funzione

2.3 Gestione degli errori semantici

È stato scelto di implementare la classe `SemanticException` per la gestione degli errori semantici. Nel caso in cui venga rilevato un errore, viene lanciata un'eccezione, che provoca l'interruzione immediata del programma.
In particolare, ogni sottoclasse che estende la superclasse `SemanticException` cattura uno specifico errore:

- `IdNotDefinedException` è sollevata nel caso in cui una variabile o una funzione venga utilizzata in lettura o scrittura, ma la sua dichiarazione non è presente nella tabella dei simboli.

- `IdAlreadyDefinedException` è sollevata nel caso in cui l'identificatore di una variabile o di una funzione è già presente all'interno dello stesso scope. In questo modo si evitano le dichiarazioni multiple.

- `TypeCheckingException` è sollevata nel caso in cui non vi è alcuna corrispondenza tra il tipo trovato e il tipo previsto. In particolare, gli errori presi in considerazione durante la fase di type checking possono essere di diversa natura. Ad esempio, in un assegnamento viene controllato che il tipo della variabile corrisponda a quello dell'espressione assegnata. Nel caso di una chiamata di funzione, invece, viene controllato

che il `TypeValue` dell'identificatore corrisponda a una funzione e che i tipi dei parametri attuali corrispondano a quelli dei parametri formali. Una dichiarazione di funzione supera la fase di type checking se vi è incompatibilità tra il suo tipo di ritorno e il tipo del corpo della funzione. Nel caso del comando `delete`, viene verificato che la variabile cancellata sia di tipo puntatore. Infine, non è consentito effettuare una `print` di puntatori o di funzioni, ma è permesso solo stampare espressioni di tipo intero o booleano.

- `EffectsException` sollevata nel caso in cui la fase di analisi degli effetti fallisca. È stata posta particolare attenzione ai fenomeni di doppia cancellazione e aliasing.

3 Sistema dei tipi

Un sistema di tipi ha il compito di associare, a tempo di compilazione, un tipo a un valore computato e di controllare che non vi siano operazioni tra tipi non compatibili fra loro, evitando cioè *errori di tipo*.
In questa sezione verrà discusso il sistema dei tipi implementato per `SimpLan Plus`. In sez. 3.1 vengono discussi i tipi computati dal linguaggio. In sez. 3.2 vengono discusse le espressioni che riguardano valori, identificatori e puntatori. In sez. 3.3 e sez. 3.4 vengono mostrate le regole di inferenza delle operazioni che lavorano rispettivamente con valori interi e booleani. In sez. 3.5 vengono discusse le regole di inferenza per gli operatori di confronto. In sez. 3.6 e sez. 3.7 vengono discussi gli *statement* e le chiamate di funzione. In conclusione, in sez. 3.8 vengono mostrate le regole di inferenza per le dichiarazioni.

3.1 Tipi

$$\frac{\Gamma\ is\ well-formed}{\Gamma \vdash int : Integer}\ \text{(T-IntType)}$$

$$\frac{\Gamma\ is\ well-formed}{\Gamma \vdash bool : Bool}\ \text{(T-BoolType)}$$

$$\frac{\Gamma\ is\ well-formed}{\Gamma \vdash \hat{}T : Pointer\ T}\ \text{(T-PointerType)}$$

I valori computati da `SimpLan Plus` possono avere tipo `Integer`, `Bool`, `Void` o `Pointer T`. Le regole di inferenza T-IntType e T-BoolType valutano un nodo `TypeNode` e lo computano al tipo corrispondente. Il valore `Void`, invece, non viene mai direttamente assegnato, ma può essere definito come valore di ritorno di una funzione. Infine, è interessante notare la presenza di un tipo ricorsivo *puntatore*, denotato dalla regola T-PointerType.

3.2 Espressioni

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n\ : Integer}\ \text{(T-IntegerExp)}$$

$$\frac{b \in \{true, false\}}{\Gamma \vdash b\ : Bool}\ \text{(T-BoolExp)}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash\ x\ : T}\ \text{(T-Lhs-Id)}$$

$$\frac{\Gamma(x) = \ Pointer\ T}{\Gamma \vdash \ x\hat{}: T} \text{ (T-Lhs-Pointer)}$$

$$\frac{\Gamma \vdash T : T_1}{\Gamma \vdash new\ T\ : Pointer\ T_1} \text{ (T-NewExp)}$$

$$\frac{\Gamma \vdash e_1 : T}{\Gamma \vdash (e_1)\ : T} \text{ (T-BaseExp)}$$

In questa sezione vengono mostrate le regole relative alle espressioni dei casi base. T-IntegerExp e T-BoolExp mostrano le regole di inferenze dei valori primitivi. T-Lhs-Id e T-Lhs-Pointer sono le regole di inferenze relative a identificatori e alla dereferenziazione dei puntatori. T-NewExp è la regola di inferenza per l'assegnamento di una nuova cella di memoria. Per concludere, T-BaseExp è la regola di inferenza per una espressione chiusa tra parentesi.

3.3 Espressioni su interi

$$\frac{\Gamma \vdash e_1 : Integer}{\Gamma \vdash -e_1\ : Integer} \text{ (T-NegExp)}$$

$$\frac{\Gamma \vdash e_1 : Integer \qquad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 + e_2\ : Integer} \text{ (T-Add)}$$

$$\frac{\Gamma \vdash e_1 : Integer \qquad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 - e_2\ : Integer} \text{ (T-Sub)}$$

$$\frac{\Gamma \vdash e_1 : Integer \qquad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 * e_2\ : Integer} \text{ (T-Mul)}$$

$$\frac{\Gamma \vdash e_1 : Integer \qquad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1/e_2\ : Integer} \text{ (T-Div)}$$

Queste regole fanno riferimento alle espressioni che operano tra interi. Le regole T-Add, T-Sub, T-Mul e T-Div sono praticamente identiche tra loro, unica differenza l'operatore. T-NegExp è invece l'operatore per il cambio di segno.

3.4 Espressioni su booleani

$$\frac{\Gamma \vdash e_1 : Bool}{\Gamma \vdash !\, e_1 \ : Bool} \text{ (T-NotExp)}$$

$$\frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1 \&\& e_2 \ : Bool} \text{ (T-AndExp)}$$

$$\frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1 || e_2 \ : Bool} \text{ (T-OrExp)}$$

Queste regole fanno riferimento alle espressioni che operano tra booleani. T-AndExp e T-OrExp fanno riferimento alle classiche operazioni di *and* logico e *or* logico. T-NotExp è la classica operazione di *not* logico.

3.5 Espressioni di confronto

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 == e_2 \ : Bool} \text{ (T-Eq)}$$

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 != e_2 \ : Bool} \text{ (T-NotEq)}$$

$$\frac{\Gamma \vdash e_1 : Integer \qquad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 > e_2 \ : Bool} \text{ (T-More)}$$

$$\frac{\Gamma \vdash e_1 : Integer \qquad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 < e_2 \ : Bool} \text{ (T-Less)}$$

$$\frac{\Gamma \vdash e_1 : Integer \qquad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 >= e_2 \ : Bool} \text{ (T-MoreEq)}$$

$$\frac{\Gamma \vdash e_1 : Integer \qquad \Gamma \vdash e_2 : Integer}{\Gamma \vdash e_1 <= e_2 \ : Bool} \text{ (T-LessEq)}$$

Gli operatori di uguaglianza e disuguaglianza, descritti dalle regole di inferenza T-Eq e T-NotEq, ricevono in input due espressioni polimorfe - che hanno il vincolo di avere tipo uguale tra loro - e restituiscono un valore booleano. Invece, le regole di inferenza T-More, T-Less, T-MoreEq e T-LessEq ricevono in input due valori interi e restituiscono un booleano.

3.6 Chiamata di funzione

$$\frac{\Gamma \vdash f : (T_1 \times T_2 \times ... \times T_n) \to T_r \qquad (\Gamma \vdash e_i : T_i^e)^{i \in 1...n} \qquad (T_i = T_i^e)^{i \in 1...n}}{\Gamma \vdash \ f(e_1, e_2, ..., e_n); : T_r} \text{ (T-Call)}$$

La regola di inferenza della chiamata di funzione controlla il tipo in memoria della funzione `f`. Successivamente, verifica che l'*n-esimo* parametro della funzione `f` abbia un tipo compatibile con quello dell'*n-esima* espressione passata. Se tutti i valori coincidono, la valutazione non solleva nessun errore.

3.7 Statement

$$\frac{\Gamma \vdash lhs : T_1 \qquad \Gamma \vdash e : T_2 \qquad T_1 = T_2}{\Gamma \vdash lhs \ = \ e; \ : Void} \text{ (T-Ass)}$$

$$\frac{\Gamma \vdash e : T \qquad T \in \{Integer, Bool\}}{\Gamma \vdash print(e); \ : Void} \text{ (T-Print)}$$

$$\frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash stm1 : T_1 \qquad \Gamma \vdash stm1 : T_2 \qquad T_1 = T_2}{\Gamma \vdash if(e_1) \ stm1 \ else \ stm2 \ : Void} \text{ (T-IfNode)}$$

$$\frac{\Gamma \cdot [\,] \vdash Dec : \Gamma_1 \qquad \Gamma_1 \vdash Stm : T}{\Gamma \vdash \ \{ \ Dec \ Stm \ \} \ : Void} \text{ (T-Block)}$$

$$\frac{\Gamma(id) = Pointer \ T}{\Gamma \vdash delete \ id; \ : Void} \text{ (T-Delete)}$$

$$\frac{\Gamma \vdash e_1 : T}{\Gamma \vdash return \ \ e_1 \ \ : T} \text{ (T-Return)}$$

Gli statement, dopo aver valutato la premessa, restituiscono il tipo `Void`. L'unica eccezione si verifica con il T-Return che, invece, dopo aver valutato la premessa, restituisce il tipo dell'espressione valutata.

3.8 Dichiarazioni

$$\frac{\Gamma \vdash T : T_1 \qquad \Gamma \vdash e : T_2 \qquad T_1 = T_2 \qquad x \notin Top(\Gamma)}{\Gamma \vdash T \ x = e; \ : \Gamma[x \to T_1]} \text{ (T-DecVar)}$$

$$\frac{\Gamma \; is \; well - formed}{\Gamma \vdash T \; id : T} \; \text{(T-Arg)}$$

$$\frac{\begin{array}{c}\Gamma \vdash T : T_r \qquad (\Gamma \vdash arg_i : T_i)^{i \in 1...n} \qquad \Gamma_1 \; contains \; only \; functions \; in \; \Gamma \\ \Gamma_2 = \Gamma_1 \cdot [arg_1.id \to T_1, ..., \; arg_n.id \to T_n, \; f \to (T_1 \times ... \times T_n) \to T_r \\ \Gamma_2 \vdash body : Void \qquad \forall \; return \; e_n \in block \implies (\Gamma_2 \vdash e_n : T_i) \wedge (T_r = T_i)\end{array}}{\Gamma \vdash T \; f(arg_1, ..., arg_n) \; body \; : \Gamma[f \to (T_1 \times ... \times T_n) \to T_r]} \; \text{(T-DecFun)}$$

Per quanto riguarda la regola di inferenza T-DecVar viene valutato semplicemente se il tipo dell'espressione e sia compatibile con il tipo T (i valori possibili sono descritti in sez. 3.1), se il controllo va a buon fine la coppia *(id, valore)* viene aggiunta alla memoria Γ. La regola T-Arg, invece, computa solamente il tipo dell'argomento passato alla funzione.
Molto interessante è, invece, il comportamento della regola T-DecFun. Per prima cosa viene computato il valore di ritorno della funzione e, successivamente, vengono computati i tipi degli argomenti della funzione. A questo punto viene costruito un nuovo ambiente Γ_1 che contiene solamente le funzioni definite precedentemente (non è possibile accedere a variabili globali dentro le funzioni). Si prosegue con la valutazione del corpo della funzione `f`, questa viene fatta sull'ambiente Γ_1 concatenato a un nuovo ambiente a cui viene aggiunto il tipo della funzione e le coppie $(id, tipo)$ dei parametri. Per controllare che il tipo restituito dalla funzione sia corretto viene fatta una ricerca in profondità degli statement `return` e viene controllato che il tipo di tutti i `return` sia compatibile con il tipo inferito da T. In conclusione, viene fatto il *binding* in Γ tra `f` e il tipo della funzione.

4 Analisi degli effetti

L'obiettivo della fase di analisi degli effetti è quello di catturare eventuali errori non rilevati nelle fasi precedenti, relativi allo scorretto utilizzo di particolari operazioni.

4.1 Regole di inferenza

In questa sezione vengono mostrate le regole di inferenza dei principali costrutti utilizzate nell'analisi degli effetti:

$$\frac{Ids(e) = \{x_1 ... x_n\}}{\Sigma \vdash \ e \ : \Sigma \triangleright [x_1 \to rw, ..., x_n \to rw]} \text{ (Eff-Exp)}$$

$$\frac{\Sigma \vdash \ e : \Sigma' \qquad \Sigma(x) \leq rw}{\Sigma \vdash \ x = e; : \Sigma' \triangleright [x \to rw]} \text{ (Eff-Assgn)}$$

$$\frac{\Sigma(x) \leq rw}{\Sigma \vdash \ delete\ x; : \Sigma \triangleright [x \to delete]} \text{ (Eff-Del)}$$

$$\frac{\Sigma \vdash \ e : \Sigma' \qquad \Sigma' \vdash s_1 : \Sigma_1 \qquad \Sigma' \vdash s_2 : \Sigma_2}{\Sigma \vdash \ if(e)\{s_1\}else\{s_2\} : max(\Sigma_1, \Sigma_2)} \text{ (Eff-If)}$$

$$\frac{}{\Sigma \vdash \ Tx; : \Sigma[x \to bottom]} \text{ (Eff-DecVar)}$$

$$\frac{\Sigma \cdot [\,] \vdash D : \Sigma' \qquad \Sigma' \vdash S : \Sigma''}{\Sigma \vdash \ D \quad S; : \Sigma''} \text{ (Eff-Block)}$$

$$\frac{\Sigma(x) \leq delete}{\Sigma \vdash \ x : \Sigma \triangleright [x \to rw]} \text{ (Eff-Lhs-Id)}$$

$$\frac{\Sigma(x) \leq delete}{\Sigma \vdash x\hat{} : \Sigma \triangleright [x \to rw]} \text{ (Eff-Lhs-Pointer)}$$

$$\frac{\begin{array}{c}\Sigma_0 = [x_1 \to \bot, ..., x_n \to \bot] \qquad \Sigma_0[f \to \Sigma_1] \vdash S : \ \Sigma \cdot \Sigma_1[f \to \Sigma_1] \\ \forall\, i > 0 \ \wedge \ \Sigma_i \ \neq \ \Sigma_{i+1} \implies \ \Sigma_i[f \to \Sigma_{i+1}] \vdash S : \ \Sigma \cdot \Sigma_{i+1}[f \to \Sigma_{i+1}]\end{array}}{\Sigma \vdash \ f(T_1 x_1, ..., T_n x_n) \quad S; : \Sigma[f \to \Sigma_{i+1}]} \text{ (Eff-FunR)}$$

$$\frac{\Sigma(f) = \Sigma_1 \qquad s_i = \Sigma(u_i) \triangleright \Sigma_1(x_i) \qquad if(u_i \notin \Sigma_2)\ then\ \Sigma_2[u_i \to s_i]\ else\ \Sigma_2[u_i \to \Sigma_2(u_i) \otimes s_i]}{\Sigma \vdash\ f(u_1, ..., u_m,\ e_1, ..., e_n);\ : \Sigma[u_i \to \Sigma_2(u_i)]}\ (\textsc{Eff-Call})$$

Le definizioni degli operatori `max, par` e `seq` che sono state utilizzate nella fase di analisi degli effetti, sono quelle studiate durante il corso.

4.2 Dettagli implementativi per parametri passati per riferimento

L'attenzione è stata posta sul cambiamento di stato delle variabili registrate nell'ambiente, per evitare fenomeni che potrebbero alterare la corretta esecuzione del programma.
In particolare, è stato considerato il fenomeno dell'aliasing nel caso di variabili passate per riferimento a una funzione, ovvero la situazione in cui la stessa variabile puntatore viene associata a più parametri formali diversi. Questa situazione può provocare effetti laterali che è necessario evitare per assicurare la correttezza del programma. La soluzione che è stata proposta prevede il controllo dei tipi effetto dei parametri attuali registrati nell'ambiente fino all'istruzione immediatamente precedente alla chiamata di funzione con i tipi effetto dei corrispondenti parametri formali calcolati al termine del corpo della funzione.
Si assuma che $u_1 ... u_n$ siano i parametri attuali passati per riferimento e Σ_2 un nuovo ambiente temporaneo per memorizzare il calcolo finale del tipo effetto che risolve definitivamente il problema dell'aliasing. Nel caso in cui l'identificatore del parametro attuale u_i non sia presente viene calcolato il tipo effetto s_i con l'operatore `seq` (eq. 1) e si aggiorna Σ_2 (eq. 2). Invece, nel caso in cui l'identificatore del parametro attuale u_i, passato per riferimento, sia presente in Σ_2, prima viene calcolato il tipo effetto s_i, attraverso l'operatore di sequenza `seq` (eq. 1), successivamente, attraverso l'utilizzo dell'operatore di parallelismo `par`, si calcola il tipo effetto finale per u_i (eq. 3).

$$s_i = \Sigma(u_i) \triangleright \Sigma_1(x_i) \tag{1}$$

$$\Sigma_2(u_i) = \Sigma_2[u_i \to s_i] \tag{2}$$

$$\Sigma_2(u_i) = \Sigma_2(u_i) \otimes s_i \tag{3}$$

Inoltre, è stata posta attenzione ai tipi effetto delle variabili passate per riferimento nel caso in cui vengano cancellate all'interno del corpo della funzione. Infatti, se nelle istruzioni successive alla chiamata, tali variabili vengono accedute in lettura o scrittura, è necessario rilevare un errore.

Siano Σ l'ambiente corrente che contiene la dichiarazione della funzione f, con $\Sigma(f) : \Sigma_0 \rightarrow \Sigma_1$ e $u_1 ... u_n$ i parametri attuali passati per riferimento alla funzione f. La soluzione proposta prevede l'aggiornamento dell'ambiente esterno $\Sigma(u_i)$ con i tipi effetto delle variabili passate per riferimento memorizzati nel codominio di *f*. La definizione formale del vincolo necessario per evitare effetti laterali dovuti al problema precedentemente descritto è la seguente:

$$\Sigma(u_i) \leq Delete \tag{4}$$

5 Generazione del codice

Lo scopo di questa fase è generare il bytecode, ovvero un set di istruzioni progettato per essere eseguito da un interprete.
Per supportare la gestione della memoria sono state utilizzate alcune variabili, ognuna delle quali è necessaria per il corretto funzionamento del programma:

- `fp`: il *frame pointer* memorizza l'indirizzo della cella di memoria occupata dall'ultima variabile inserita e permette di accedere alle celle sottostanti attraverso l'offset.
- `sp`: lo *stack pointer* memorizza l'indirizzo dell'ultima cella occupata dello stack.
- `hp`: l'*heap pointer* è necessario per la gestione delle variabili puntatore, infatti tiene traccia dell'ultima cella occupata nell'heap che contiene l'ultimo oggetto puntato che è stato memorizzato.
- `ip`: l'*istruction pointer* memorizza l'istruzione correntemente eseguita, sulla base del codice.

Inoltre, per consentire ai programmi di eseguire correttamente funzioni, anche ricorsive, è stato necessario memorizzare in opportune variabili due principali informazioni:

- `rv`: valore di ritorno delle funzioni, necessario solo nel caso in cui il tipo di ritorno della funzione sia `int`, `bool` o `pointer`, ciò significa che nel corpo della funzione è presente un comando 'return exp'.
- `ra`: l'indirizzo di ritorno delle funzioni, fondamentale per consentire al programma di tornare all'istruzione successiva alla chiamata di funzione, al termine del corpo.

Il bytecode che è stato utilizzato è costituito dalle seguenti istruzioni:

- `PUSH`: memorizza sulla pila il valore specificato e decrementa lo stack pointer.
- `POP`: rimuove l'ultimo elemento della pila e incrementa lo stack pointer.
- `ADD`: somma e rimuove i primi due valori memorizzati sulla pila e inserisce il risultato in testa.
- `MULT`: moltiplica e rimuove i primi due valori in pila e inserisce il risultato in testa.
- `DIV`: divide il secondo valore in pila per il primo, rimuove i primi due valori in testa e inserisce il risultato.
- `SUB`: sottrae il secondo valore in pila al primo, rimuove i primi due valori in testa e inserisce il risultato.

- `AND`: inserisce in testa alla pila il valore 1 se i primi due valori in pila sono diversi da 0, altrimenti inserisce 0.
- `OR` inserisce in testa alla pila il valore 1 se almeno uno dei primi due valori in pila è diverso da 0, altrimenti inserisce 0.
- `NOT`: se il valore in testa alla pila è pari a 1 lo sostituisce con il valore 0 e viceversa.
- `STOREW`: rimuove dalla testa della pila il valore che rappresenta l'indirizzo, rimuove il secondo valore dalla pila e lo memorizza nell'indirizzo individuato precedentemente.
- `LOADW`: memorizza sulla pila il valore contenuto nella cella di memoria con indirizzo corrispondente al valore presente nella testa dello stack.
- `BRANCH`: salta all'istruzione corrispondente alla label specificata.
- `BRANCHFUN`: salta all'istruzione corrispondente alla label della definizione di funzione e aggiorna il return address con il valore dell'*istruction pointer*.
- `BRANCHEQ`: salta all'istruzione corrispondente alla label specificata se i primi due elementi in testa alla pila sono uguali.
- `BRANCHNEQ`: salta all'istruzione corrispondente alla label specificata se i primi due elementi in testa alla pila sono diversi.
- `EQ`: se i primi due valori della pila sono uguali inserisce in testa alla pila il valore 1, altrimenti il valore 0.
- `NEQ`: se i primi due valori della pila sono diversi inserisce in testa alla pila il valore 1 , altrimenti il valore 0.
- `LESSOP`: inserisce in testa alla pila il valore 1 se il secondo valore della pila è minore del primo.
- `LESSEQ`: inserisce in testa alla pila il valore 1 se il secondo valore nella pila è minore o uguale al primo.
- `MOREOP`: inserisce in testa alla pila il valore 1 se il secondo valore nella pila è maggiore del primo.
- `MOREEQ`: inserisce in testa alla pila il valore 1 se il secondo valore nella pila è maggiore o uguale al primo.
- `JS`: salta all'istruzione immediatamente successiva all'ultimo valore presente sulla pila.
- `STORERA`: inserisce in testa alla pila il return address.

- `LOADRA`: rimuove il valore in testa alla pila e lo memorizza nel return address.
- `STORERV`: inserisce in testa alla pila il return value.
- `LOADRV`: rimuove il valore in testa alla pila e lo memorizza nel return value.
- `STOREFP`: inserisce in testa alla pila il frame pointer.
- `LOADFP`: rimuove il valore in testa alla pila e lo memorizza nel frame pointer.
- `COPYFP`: memorizza nel frame pointer il valore dello stack pointer.
- `STOREHP`: rimuove il valore in testa alla pila e lo memorizza nell'heap pointer.
- `LOADHP`: inserisce in testa alla pila l'heap pointer.
- `PRINT`: stampa il valore memorizzato a indirizzo sp.
- `HALT`: stampa il risultato del programma (se presente un valore di ritorno) e termina la generazione del codice.

Per generare il codice è stata utilizzata la definizione della funzione `cgen`, implementata per ogni costrutto, come mostrato nelle sottosezioni successive.
Si osservi che lo stack è invariante a meno dell'ultimo elemento. Infatti, al termine di ogni cgen è necessario che la pila non sia stata modificata a meno dell'ultimo elemento.

5.1 Programma

```
cgen(OffsetSymTable S,{D; C;}) =
    label = newlabel()
    sfp
    cfp
    cgen(S, D_var)
    sra
    b label
    cgen(S, D_fun)
    label:
    cgen(S, C)
    lra
    for(i=0; i<S.size; i++)
        pop
    lfp
```

Figura 8: funzione per la generazione del codice del blocco

5.2 Dichiarazioni

5.2.1 Dichiarazione di funzione

```
cgen(OffsetSymTable S, T fun(T1 x1,...,Tn xn ){block}) =
    label_fun:
    sra
    cgen(S, block)
    sra
    js ra
```

Figura 9: funzione per la generazione del codice di una dichiarazione di funzione

5.2.2 Dichiarazione di variabile

L'espressione tra le parentesi quadre è opzionale.

```
cgen(OffsetSymTable S, T x [= e];) =
   cgen(S, e) //se presente l'inizializzazione
   push 34 //altrimenti
```

Figura 10: funzione per la generazione del codice di una dichiarazione di variabile

5.3 Comandi

5.3.1 Condizionale

L'espressione tra le parentesi quadre è opzionale.

```
cgen(OffsetSymTable S, if (C){stm1} [else {stm2}]) =
    //condizionale con ramo else
        cgen(S, C)
        push 1
        beq label1
        cgen(S, stm2)
        b label2
        label1:
        cgen(S, stm1)
        label2:
    //condizionale senza ramo else
        cgen(S, C)
        push 1
        bneq label1
        cgen(S, stm1)
        label1:
```

Figura 11: funzione per la generazione del codice del comando condizionale

5.3.2 Assegnamento

```
cgen(OffsetSymTable S, id = e; ) =
    cgen(S,e)
    sfp
    if(id is not a pointer)
        for(i=0; i<nestingLevel-S(id).nestingLevel; i++)
             lw
    push S(id).offset
    sub
    if (id is pointer)
        for(i=0; i< # puntatori; i++)
           lw
    sw
```

Figura 12: funzione per la generazione del codice di una dichiarazione di funzione

5.3.3 Chiamata di funzione

```
cgen(OffsetSymTable S, fun(e1,...,en)) =
    sfp
    cfp
    for (i=0; i<n; i++) {
        cgen(S, ei)
    }
    bf label_fun
    lra
    for (i=0; i<n; i++) {
        pop
    }
    lfp
```

Figura 13: funzione per la generazione del codice della chiamata di funzione

5.3.4 Print

```
cgen(OffsetSymTable S, print exp;) =
    cgen(S, exp)
    print
    pop
```

Figura 14: funzione per la generazione del codice del comando print

5.3.5 Return

L'espressione tra le parentesi quadre è opzionale.

```
cgen(OffsetSymTable S, return [exp];) =
    cgen(S, exp) //se return con espressione
    push 34 // se return senza espressione
    lrv
        //pulisco la pila
        //se return di un blocco interno
         lra
         for(i=0; i<S.size; i++)
            pop
         lfp
         //se return di una funzione
     sra
     //se return del programma (blocco esterno)
     srv
     halt
```

Figura 15: funzione per la generazione del codice del comando return

5.4 Espressioni

L'espressione tra le parentesi quadre è opzionale.

```
cgen(OffsetSymTable S, ( e )) = cgen (S,e)
```

Figura 16: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, - e ) =
    push 0
    cgen (S, e)
    sub
```

Figura 17: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, ! e ) = cgen (S, e)
```

Figura 18: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, x[^..^] ) =
    // se lhs = x
    cgen (S, x)
    // altrimenti
    cgen (s, x^[^..^])
```

Figura 19: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, new type) =
    lhp
```

Figura 20: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 == exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   eq
```

Figura 21: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 != exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   neq
```

Figura 22: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 && exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   and
```

Figura 23: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 || exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   or
```

Figura 24: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 + exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   add
```

Figura 25: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 - exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   sub
```

Figura 26: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 * exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   mult
```

Figura 27: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 / exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   div
```

Figura 28: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 > exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   moreop
```

Figura 29: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 < exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   lessop
```

Figura 30: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 >= exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   moreeq
```

Figura 31: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, exp1 <= exp2) =
   cgen(S, exp1)
   cgen(S, exp2)
   lesseq
```

Figura 32: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, fun (e_1,..,e_n)) =
    cgen (S, CallNode(fun (e_1,..,e_n)))
    push rv
```

Figura 33: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, fun (e_1,..,e_n)) =
    cgen (S, CallNode(fun (e_1,..,e_n)))
    push rv
```

Figura 34: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, n) =
    push n
```

Figura 35: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, true) =
    push 1
```

Figura 36: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, false) =
    push 0
```

Figura 37: funzione per la generazione del codice delle espressioni

```
cgen(OffsetSymTable S, n) =
    push n
```

Figura 38: funzione per la generazione del codice delle espressioni

5.5 Lhs

L'espressione tra le parentesi quadre è opzionale.

```
cgen(OffsetSymTable S, x ) =
    sfp
    push S(x).offset
    sub
    lw
```

Figura 39: funzione per la generazione del codice delle variabili

```
cgen(OffsetSymTable S, x^[^...^] ) =
    sfp
    for(i=0; i < # puntatori; i++ )
        lw
    push S(x).offset
    sub
    lw
```

Figura 40: funzione per la generazione del codice dei puntatori

6 Testing

In questa sezione viene discussa la parte relativa ai test creati appositamente per validare il funzionamento del progetto `SimpLan Plus`. I file utilizzati si trovano nella cartella `Test/`.
Per quanto riguarda il *testing* della fase di *typechecking* e di *analisi degli effetti* è stato usato `JUnit 5`. Invece, per quanto riguarda il *testing* della generazione del codice è stato usato un programma `Java`, in cui è possibile cambiare il file da eseguire per verificare uno specifico *input*.
In sez. 6.1 viene discussa la parte relativa al testing della fase di *typechecking*. In sez. 6.2 viene, invece, discussa la parte relativa all'*analisi degli effetti*. In conclusione, in sez. 6.3 viene presentata la parte del *testing* relativa alla generazione del codice.

6.1 Typechecking

Gli *unit test* relativi al *typechecking* sono disponibili nel file `Java` posizionato al percorso `src/test/.../MainTest`. Per maggiori informazioni riguardo alle regole di typechecking si fa riferimento alla sez. 3. Il totale degli esempi testati tramite *unit test* è 29.
I primi nove test sono quelli relativi alla specifica, invece, i successivi test sono stati realizzati per valutare in modo specifico il comportamento del compilatore.

```
{
    int u = 1 ;
    void f(^int x, int n){
            if (n == 0) { print(x) ; delete x ; }
            else { ^int y = new int; y^ = x^ * n ; f(y,n-1) ; }
    }
    f(u,6);
}
```

Figura 41: Esempio di programma che cerca di stampare il valore di un puntatore

Viene riportato un esempio di *test* in Figura 41. L'esempio mostrato solleva un errore a tempo di typechecking (Figura 42) ed è causato dal tentativo di stampare un valore di tipo puntatore.
Un altro esempio viene mostrato in Figura 43. In questo caso si sta cercando di eseguire l'operazione di `delete` su un valore non di tipo puntatore, questa operazione non è concessa. Il risultato della compilazione è mostrato in Figura 44.

```
Exception in thread "main" Utils.SemException.TypeCheckingException:
        Il valore nella print non è valido
Tipo previsto:
        TypeExp{type=INTEGER, ptr='null'}, return='null'}, list=''}
Tipo trovato:
        TypeExp{type=POINTER, ptr='TypeExp{type=INTEGER, ptr='null'},
        ↪  return='null'}, list='}'}, return='null'}, list=''}
```

Figura 42: Errore sollevato a tempo di typechecking da Figura 41

```
    {
        int x;
        delete x;
        x = 10;
    }
```

Figura 43: Esempio di programma che cerca di eliminare un valore non di tipo puntatore

```
Exception in thread "main" Utils.SemException.TypeCheckingException:
    Errore nella delete
Tipo previsto: TypeExp{type=POINTER, ptr=''null'}, return=''null'},
↪  list=''}
Tipo trovato: TypeExp{type=INTEGER, ptr=''null'}, return=''null'},
↪  list=''}
```

Figura 44: Errore sollevato a tempo di typechecking da Figura 43

6.2 Analisi degli effetti

Per quanto riguarda la parte di *testing* dell'*analisi degli effetti* vale quanto detto nella sezione precedente. Gli *unit test* relativi a questa parte sono disponibili nel file `Java` posizionato al percorso `src/test/.../MainTest`. Il totale degli esempi testati tramite *unit test* è 29. Inoltre, sono stati realizzati dei test mirati per controllare che non avvengano fenomeni quali

p.e. *aliasing* o l'accesso a zone di memoria rimosse.
In Figura 45 viene mostrato un esempio di programma che cerca di eliminare due volte la stessa cella di memoria. Il risultato della compilazione viene mostrato in Figura 46. Il programma dichiara una funzione il cui unico compito è quello di cancellare una cella di memoria e, successivamente, chiama questa funzione due volte sulla stessa variabile. L'obiettivo di questo esempio è quello di mostrare come il fenomeno dell'*aliasing* sia gestito in modo corretto dal punto di vista del passaggio dei parametri.

```
{
    ^int x;

    void g(^int y){
        delete y;
    }

    g(x);
    g(x);
}
```

Figura 45: Esempio di programma che cerca di eliminare due volte la stessa cella di memoria

```
Exception in thread "main" Utils.SemException.EffectsException:
    Errore nella chiamata di funzione
Valore precedente: DELETE
Valore successivo: TOP
```

Figura 46: Risultato della compilazione di Figura 45 e di Figura 47

Allo stesso modo, il *test* mostrato in Figura 47 contiene un programma che cerca di accedere a una cella di memoria eliminata, e prova a farlo grazie al fenomeno dell'*aliasing*. Il risultato della compilazione è uguale al *test* precedente e viene mostrato in Figura 46. Il programma dichiara una funzione che prende in *input* tre parametri: il primo viene solamente eliminato, il secondo e il terzo modificati. A questo punto la funzione viene chiamata passando due volte lo stesso parametro. Il problema che emerge in questo caso è l'accesso in scrittura a una variabile precedentemente eliminata. Il compilatore riconosce

l'errore nel programma e segnala un *upgrade* non concesso.

```
{
    void f(^int x, ^int y, int z){
        delete x;
        y^=3;
        z=5;
    }

    ^int x;
    int z;

    f(x,x,z);
}
```

Figura 47: Esempio di programma che cerca di accedere a una cella di memoria eliminata

6.3 Generazione del codice

Per quanto riguarda il *testing* della generazione del codice è stato usato un programma `Java`, in cui è possibile cambiare il file da eseguire per verificare uno specifico *input*. Il file è posizionato al percorso `src/main/java/.../Main`. Sono stati testati sia i programmi presenti negli *unit test* discussi prima sia dei programmi realizzati *ad hoc* per valutare il comportamento di questa fase.

```
{
    int f(int x){
        if(x==0) return 0;
        if(x==1 || x==2) return 1;
        return f(x-1) + f(x-2);
    }
    return f(15);
}
```

Figura 48: Esempio di programma in `SimpLan Plus` che calcola *Fibonacci*

In Figura 48 viene mostrato il *test* contenuto nel file `Test/test-fibonacci.plan`. L'obiettivo di questo programma è quello di calcolare il $15-esimo$ numero della sequenza di *Fibonacci*. Il risultato dell'esecuzione del programma è mostrato in Figura 49. Come si può vedere dalla figura il numero calcolato è corretto, questo dimostra come il codice compilato riesca a gestire correttamente anche le funzioni ricorsive. Nella lista dei *test* è presente anche un esempio di *Fibonacci* con i puntatori e un esempio di calcolo del *Fattoriale*.

```
Starting Virtual Machine...

Result: 610

Finishing Virtual Machine...
```

Figura 49: Risultato esecuzione *Fibonacci*

```
Starting Virtual Machine...
3
10
24
Finishing Virtual Machine...
```

Figura 50: Risultato esecuzione Figura 51 a pagina successiva

Il test mostrato in Figura 51 verifica il funzionamento delle funzioni annidate. Questo esempio, inoltre, controlla il riuso degli identificatori all'interno di tali funzioni. Il programma dichiara per prima cosa una funzione `g`, il cui compito è quello di stampare la variabile in *input*. Successivamente, viene dichiarata una funzione `f` che stampa il risultato di `g` (funzione dichiarata internamente), che calcola la sommatoria da 1 a n. Il programma continua con la dichiarazione della funzione `stampa`, che al suo interno chiama una funzione `g`, che calcola il fattoriale del numero in *input*. A questo punto vengono chiamate in ordine le tre funzioni `g`, `f` e `stampa`. Il risultato è mostrato in Figura 50.

```
{
    void g(int x){
        print(x);
    }

    void f(int x){
        int g(int x){
            if(x == 0){
                return 0;
            }
            return x + g(x - 1);
        }
        print(g(x));
    }

    void stampa(int x){
        int g(int x){
            if(x == 0){
                return 1;
            }
            return x * g(x-1);
        }
        print(g(x));
    }
    g(3);
    f(4);
    stampa(4);
}
```

Figura 51: Esempio di programma che usa funzioni di tipo *nested*

Riferimenti bibliografici

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Series in Computer Science / World Student Series Edition. Addison-Wesley, 1986.

[GM10] Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Undergraduate Topics in Computer Science. Springer, 2010.

[Mit90] John C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 365–458. Elsevier and MIT Press, 1990.

[Mog17] Torben Ægidius Mogensen. *Introduction to Compiler Design, Second Edition*. Undergraduate Topics in Computer Science. Springer, 2017.

[WS10] Reinhard Wilhelm and Helmut Seidl. *Compiler Design - Virtual Machines*. Springer, 2010.