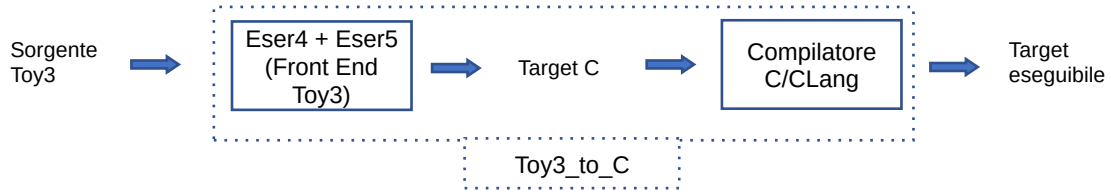
ESERCITAZIONE 5

Costruire un analizzatore semantico per Toy3 facendo riferimento alle informazioni di cui sotto e legandolo ai due analizzatori già prodotti nell'esercizio 4.

Dopo la fase di analisi semantica si sviluppi inoltre un ulteriore visitor del nuovo AST che produca la traduzione **in linguaggio C** (versione Clang) di un sorgente Toy3.

In questa esercitazione, bisogna quindi produrre un compilatore completo che prenda in input un codice Toy3 e lo compili in un programma C.
Il programma C risultante deve essere compilabile tramite C senza errori*, e deve eseguire correttamente (ovvero in modo equivalente a quanto atteso dal programma Toy3 sorgente).*
Produrre quindi un unico script ***Toy3_to_C*** che metta insieme i due moduli (Toy3 e C) e che, lanciato da linea di comando, compili il vostro programma Toy3 in un codice eseguibile, come mostrato nella figura seguente:



Per testare Toy3_*to_C*, oltre ad utilizzare il programma sviluppato nell'esercizio 4, si sviluppino programma Toy3 di test.

(Esempi di codice C, da cui trarre spunto, per ciascuno dei problemi citati sopra li trovate qui: https://person.dibris.unige.it/reggio-gianna/LPWWW00/LEZIONI/PARTE3/ESEMPI.html)

CONSEGNA
Tutto il codice e i files di test utilizzati per testare il proprio compilatore vanno prodotti come al solito su GitLab. Sulla piattaforma elearning va consegnato il link al progetto ed anche un documento che descriva tutte le scelte effettuate durante lo sviluppo del compilatore che si discostano dalle specifiche date o che non sono presenti nelle specifiche.

Analisi Semantica di Toy3

(Prima di leggere questa traccia si studi bene il contenuto delle lezioni riguardanti l'analisi semantica.)

L'analisi semantica deve svolgere almeno i seguenti compiti:

Gestione dello scoping: questo compito crea la tabella dei simboli a partire dalle dichiarazioni contenute nel programma tenendo conto degli scope. Esempi di regole di scoping da rispettare di solito indicano che gli identificatori devono essere dichiarati (prima o dopo il loro uso), che un identificatore non deve essere dichiarato più volte nello stesso scoping, etc., Di solito la prima cosa da fare è **individuare quali sono i costrutti del linguaggio che individuano un nuovo scoping** (e quindi devono far partire la costruzione di una nuova tabella). Ad esempio, in Java i costrutti class, method indicano scoping specifici. Nel nostro caso ci sono vari costrutti[1] che portano alla costruzione di un nuovo scope: La parte di programma fra le parole chiavi *program* e *begin* (per le variabili globali e le funzioni), quella fra

[1] **Un costrutto del linguaggio avrà sempre un nodo corrispondente nell'albero sintattico (AST) e una o più produzioni corrispondenti nella grammatica**

le parole chiavi *begin* e *end* (per le variabili locali del programma), la dichiarazione di funzione/procedura (per i parametri e le variabili locali), il corpo delle istruzioni 'while', 'if-then', e 'if-then-else' (due corpi in questo caso).
Per implementare la most-closely-nested rule si faccia riferimento a quanto descritto al corso ed al materiale indicato sulla piattaforma. In ogni punto del programma (oppure nodo dell'AST), la gestione dello scoping deve fornire il *type environment* relativo ad esso attivo in quel punto (ovvero la catena di tabelle dei simboli attive in quel punto).

Inferenza di tipo: il linguaggio Toy3 prevede la possibilità di dichiarare una variabile senza indicare esplicitamente il tipo ma assegnandole un valore costante (es. *x : 10*). L'analisi semantica in questo caso deve inferire il tipo intero di x dal tipo della costante 10 ed inserire (x, int) nel type environment corrente.

Type checking: utilizzando il type environment, il type checking controlla che le variabili siano dichiarate propriamente (una ed una sola volta) ed usate correttamente secondo le loro dichiarazioni, per **ogni costrutto** del linguaggio. Le regole di controllo di tipo costituiscono il "type system", sono date in fase di definizione del linguaggio e sono descritte con regole di inferenza di tipo IF-THEN. Per ogni costrutto del linguaggio una regola deve

- indicare i *tipi degli argomenti* del costrutto affinché questo possa essere eseguito.
- indicare il *tipo del costrutto* una volta noti quelli dei suoi argomenti.

I tipi degli identificatori verranno presi dal type environment associato al costrutto.
Ad esempio, dato il costrutto somma: a + b, nel type environment O, il type system conterrà la regola

"IF (a is integer in O) and (b is integer in O) THEN a+b has type integer in O",

che, in poche parole, afferma

1. che la somma è semanticamente corretta se i suoi due argomenti sono interi (ciò non esclude che non possa essere corretta anche in altri casi)
2. che la somma di due interi è ancora un intero;

oppure, per il costrutto ***chiamata a funzione*** **f(arg1, arg2)** con type environment corrente O, il type system avrà la regola

"IF (**f** è una funzione presente in O ove è descritta con argomenti di tipo **t1** e **t2** e tipo di ritorno **t**) AND (arg1 è compatibile con il tipo **t1** ed arg2 è compatibile con il tipo **t2** in O)
THEN la chiamata f(arg1, arg2) è semanticamente corretta in O ed ha tipo **t**
ELSE c'è un type mismatch"

Ad esempio, se la dichiarazione in O è f(int, double):int allora la chiamata f(1, 2.3) è

1. **ben tipata** e
2. **restituisce un intero**.

L'analisi semantica è implementata di solito tramite una o più visite dell'albero sintattico (AST) legato alla tabella delle stringhe come generato dall'analisi sintattica.
Nel caso di Toy3 le funzioni e variabili globali possono essere dichiarate anche dopo l'uso per cui si possono realizzare due visite oppure una sola visita "smart".
Tutti i controlli non fatti sintatticamente andranno fatti qui nell'analisi semantica. Ad esempio, nella chiamata a funzione/procedura vanno anche gestiti i parametri di tipo ref: i parametri attuali corrispondenti a parametri formali di tipo ref devono sempre essere identificatori di variabili e mai altro tipo di espressione.

L'output di questa fase, se il programma è ben tipato, è l'AST i cui nodi sono arricchiti con le informazioni di tipo e, per i nodi generatori di scoping, il riferimento al proprio type environment.

Per agevolare lo studente nell'implementare l'analizzatore semantico di Toy3, nel seguito si da uno schema **approssimato** che descrive quali sono le azioni principali da svolgere per ogni nodo dell'AST visitato.
Si noti che le azioni A e B riguardano la gestione dello scoping e quindi la creazione ed il riempimento della tabella dei simboli, mentre le rimanenti azioni riguardano il type checking e

usano le tabelle dei simboli solo per consultazione.
(Si legga il seguente testo consultando simultaneamente i documenti che descrivono la sintassi e la specifica dell'albero sintattico del linguaggio dati nell'esercitazione 4)

SCOPING

A.
Se il nodo dell'AST è legato ad un costrutto di *creazione di nuovo scope* (ProgramOp, DefDeclOp, e BodyOp solo se non è figlio di DefDeclOp[2])
allora
  se il nodo è visitato per la prima volta
    allora
      crea una nuova tabella, legala al nodo corrente e falla puntare alla tabella precedente (fai in modo di passare in seguito il riferimento di questa tabella a tutti i suoi figli, per il suo aggiornamento, qui si può usare alternativamente uno stack)

B.
Se il nodo è legato ad un costrutto di *dichiarazione variabile o funzione/procedura* (come ad es. VarDeclOp, ParDeclOp, DefDeclOp) **allora**
  se la *tabella corrente** contiene già la dichiarazione dell'identificatore coinvolto **allora**
    restituisci "errore di dichiarazione multipla"
  altrimenti
    aggiungi dichiarazione alla tabella

**Nota: Se si sceglie di usare lo stack la tabella corrente è quella sul top dello stack.*

In B. va gestita anche l'inferenza di tipo di cui sopra (es. *x : 10*).

TYPE-CHECK

In questa fase bisogna verificare che i costrutti siano ben tipati ed aggiungere un ***type*** a (quasi) tutti i **nodi** dell'albero (ovvero dare un tipo ad ogni costrutto del programma)

C.
Se il *nodo* nell'ASTè legato ad un *uso di un identificatore* **allora**
  metti in *current_table_ref* il riferimento alla tabella linkata dal *nodo* (passatogli dal padre o presente al top dello stack)
  Ripeti
    Ricerca (lookup) l'identificatore nella tabella riferita da *current_table_ref* e inserisci il suo riferimento in *temp_entry*.
    Se l'identificatore non è stato trovato **allora**
      current_table_ref = riferimento alla tabella precedente
      Se *current_table_ref* è nil (la lista delle tabelle è finita) **allora**
        restituisci "identificatore non dichiarato"
  fino a quando *temp_entry* non contiene la dichiarazione per l'identificatore;

  nodo.type = temp_entry.type;

  // è possibile memorizzare, nel nodo, il riferimento alla entry nella tabella oltre

[2] Si noti che per una DefDeclOP i parametri, e le variabili locali dichiarate nel suo BodyOp vanno in una unica tabella di scoping e non in due. Per cui la tabella la si crea sul nodo DefDecl ma non anche sul suo figlio BodyOp. Negli altri casi (if, while) BodyOp crea una sua tabella.

che il (o al posto del) suo tipo.

D.
Se il nodo è legato ad una costante (int_const, true, etc.) **allora**
node.type = tipo dato dalla costante

E.
Se il nodo è legato ad un costrutto riguardante operatori di espressioni o istruzioni
allora
controlla se i tipi dei nodi figli rispettano le specifiche del *type system*
Se il controllo ha avuto successo **allora** assegna al nodo il tipo indicato nel *type system*
altrimenti
restituisci "errore di tipo"

Vincoli non espressi nella sintassi ma che devono essere verificati nell'analisi semantica

Dichiarazione delle variabili (si veda regola nel Type system qui di sotto)
E' possibile dichiarare **solo una** variabile alla volta tramite **costante** e la variabile non deve essere inizializzata.
Ad Esempio
a | b | c : 10; // non è ammesso anche se la sintassi lo permette
a = 1 : 10; // non è ammesso anche se la sintassi lo permette
b | a = 1 | c : 10; // non è ammesso anche se la sintassi lo permette
a : 10; // OK

Assegnazione multipla di variabili:
1. non è possibile utilizzare funzioni nella parte destra dell'assegnazione multipla.
Esempio:
a := f(4); // è OK
a | b := 10, f(4); // non è ammesso
2. il numero di variabili alla sinistra deve essere uguale al numero di espressioni alla destra.
a | b := 10;
a | b := 10, 11, 12;
sono entrambi errati.

Chiamata di funzione/procedura
una chiamata a funzione o procedura deve rispettare il numero ed il tipo dei parametri formali.
Per i parametri formali passati per riferimento, il corrispondente parametro attuale deve essere il nome di una variabile (e non una qualsiasi espressione, anche se la sintassi lo permetterebbe).
Nel caso della sola chiamata a funzione, il suo tipo di ritorno deve essere compatibile con il resto dell'espressione da cui viene chiamata
Il corpo di una dichiarazione di funzione deve contenere almeno un return ed il suo tipo deve essere uguale al tipo di ritorno della funzione (si veda regola nel Type system qui di sotto)

TYPE SYSTEM

Il seguente è un sottoinsieme delle regole di tipo definite dal **progettista del linguaggio.** Le regole qui descritte vengono semplificate senza far riferimento al type environment O, che è dato per scontato, e usando i termini IF, THEN.

(Ricordo di leggere il seguente testo consultando simultaneamente i documenti che descrivono la sintassi e la specifica dell'albero sintattico del linguaggio dati nell'esercitazione 4)
costrutto *while*, nodo whileOp:

IF il tipo del primo nodo figlio è *Boolean* AND il secondo figlio BodyOp ha raggiunto con successo il tipo NOTYPE
THEN il *while* non ha errori di tipo ed il suo tipo finale è NOTYPE
ELSE nodewhileOp.type = error

costrutto *assegnazione*, nodo AssignOp :
IF il numero n degli identificatori nel primo figlio è uguale al numero dei valori restituiti dalle espressioni nel secondo figlio **AND** il tipo dell'i-esimo identificatore è compatibile con il tipo dell'i-esimo valore per i = 1, .., n
THEN l'assegnazione non ha errori di tipo ed il suo tipo finale è NOTYPE
ELSE nodeAssignOp.type = error

costrutto *condizionle*, nodo ifThenElseOp:
IF tipo del primo nodo figlio è *Boolean* e gli altri due figli BodyOp hanno raggiunto con successo il tipo NOTYPE
THEN non vi sono errori di tipo ed il suo tipo finale è NOTYPE
ELSE node.type = error

costrutto *operatore relazionale binario*, nodi GtOp, GeOp, etc.:
IF i tipi dei nodi figli primo e secondo sono tipi compatibili con l'operatore (**si veda tabella di compatibilità**)
THEN *nodo.type* = *Boolean*
ELSE node.type = error

costrutti *operatori binary*, nodi AddOp, MulOp, etc.:
IF i tipi dei dei due nodi figli sono tipi compatibili (**si veda tabella di compatibilità**)
THEN *nodo.type* = il tipo risultante dalla tabella
ELSE node.type = error

costrutto **return**, nodo ReturnOp
IF l'espressione figlia di ReturnOp ha tipo T **e** la funzione/procedura più vicina nel type environment corrente ha tipologia di *funzione* e tipo di ritorno T
THEN node.type = T; mark the *return_check* field in the table entry for the function.[3]
ELSE uso errato di return

Durante creazione delle tabelle di scoping

costrutto **dichiarazione di variabili**, nodo VarDeclOp
IF il tipo da assegnare è una costante di tipo T e se la lista delle variabili è di taglia 1 e se l'unica variabile non è inizializzata
THEN *aggiungi alla tabella dei simboli corrente l'unica variabile assegnandogli tipo T.*
ELSE node.type = error

che si può esprimere equivalentemente come:

IF il figlio destro di VarDeclOp è una costante di tipo T **e** la lista figlio sinistro di VarDeclOP ha taglia 1 **e** il figlio destro dell'unico VarOptInitOp è null
THEN *aggiungi alla tabella dei simboli corrente l'identificatore figlio sinistro di* VarOptInitOp *assegnandogli il tipo T.*
ELSE dichiarazione errata
Mancano in questo type system varie regole di tipo quali ad esempio quelle riguardanti ReadOp, CallOp ed altri il cui svolgimento è lasciato allo studente.

[3] Si potrebbe prevedere un campo per le dichiarazioni di funzioni nella tabella dei simboli che tenga conto se almeno un return è stato visto durante la sua analisi. Alla fine della visita si potrebbe controllare se tutte le funzioni sono state marcate. Naturalmente qualsiasi altra procedura atta allo scopo è permessa.

Nel file allegato *Toy3TypeSystem.pdf* ci sono alcune delle regole di tipo che utilizzano **regole di inferenza** al posto di regole **IF-THEN-ELSE**.