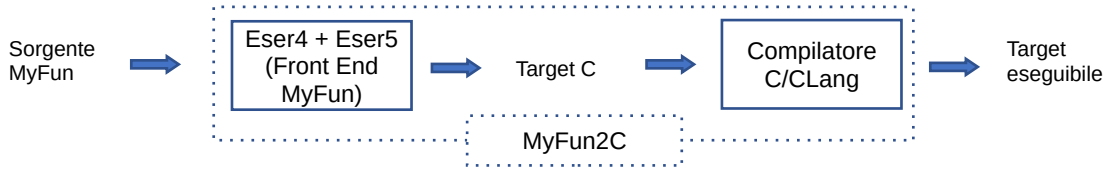
ESERCITAZIONE 5

Costruire un analizzatore semantico per MyFun 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 gcc o Clang) di un sorgente MyFun.

In questa esercitazione, bisogna quindi produrre un compilatore completo che prenda in input un codice MyFun e lo compili in un programma C.
Il programma C risultante deve essere compilabile tramite C senza errori*, e deve eseguire correttamente.*
Produrre quindi un unico script ***MyFun2C*** che metta insieme i due moduli (MyFun e C) e che, lanciato da linea di comando, compili il vostro programma MyFun in un codice eseguibile, come mostrato nella figura seguente:



Per testare MyFun*2C*, oltre ad utilizzare il programma sviluppato nell'esercizio 4, si sviluppi e compili anche il programma MyFun che svolga (a richiesta tramite menu)

1. la somma di due numeri
2. la moltiplicazione di due numeri utilizzando la somma
3. la divisione intera fra due numeri positivi
4. l'elevamento a potenza
5. la successione di Fibonacci

(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. **Il documento deve anche contenere tutte le regole di type checking effettivamente implementate nel formato regole di inferenza.**

MODALITA' DELLA VERIFICA DEL PROGETTO
Lo studente si presenta con il proprio portatile, con preistallato l'ambiente di sviluppo preferito, un compilatore C e l'intero sorgente prodotto.
Il docente richiede delle modifiche al sorgente, **lo studente farà UNA COPIA del progetto** e vi applicherà le modifiche richieste. Inoltre svilupperà, compilerà ed eseguirà uno o più **programmini di test** per mostrare la correttezza delle modifiche
In fase di verifica, quindi lo studente sarà in grado in modo agevole e semplice di lanciare il nuovo compilatore sui test prodotti (sia quelli consegnati che quelli sviluppati al momento) e mostrare le vecchie e nuove funzionalità. Quante più funzionalità verranno mostrate tramite i test tanto più verrà apprezzato il progetto. I test possono riguardare ciascuna delle fasi.

Analisi Semantica di MyFun

(Prima di leggere questa traccia si studi bene il contenuto delle lezioni 29 e 30.)

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: il programma (per le variabili globali), la dichiarazione di funzione (per i parametri e le variabili locali), il corpo main (per le sue variabili locali), lato 'then' e lato 'else' dell'istruzione 'if', il corpo dell'istruzione 'while'.
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 la parte di tabella dei simboli attiva in quel punto, altrimenti detto *type environment relativo ad esso*.

Inferenza di tipo: il linguaggio MyFun prevede la possibilità di dichiarare una variabile di tipo *var* e contestualmente assegnarle un valore costante (es. *var 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 verranno estratti dal type environment associato al costrutto.
Ad esempio, dato il costrutto somma a + b sotto type environment T, il type system conterrà la regola

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

che, in poche parole, afferma che la somma di due interi è ancora un intero;
oppure, per il costrutto ***chiamata a funzione*** **f(arg1, arg2)** il type system avrà la regola

"IF (**f** è una funzione presente in T e descritta in modo tale che prende argomenti di tipo **t1** e **t2** e restituisce un valore di tipo **t**) AND (arg1 è compatibile con il tipo **t1** ed arg2 è compatibile con il tipo **t2**)
THEN f(arg1, arg2) ha tipo **t**
ELSE c'è un type mismatch"

Ad esempio, se la dichiarazione sotto T è f(int x, double y):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 MyFun decidiamo che le funzioni sono sempre dichiarate prima dell'uso per cui una visita dovrebbe bastare.
Altri controlli possono scaturire da come avete sviluppato la grammatica. Controlli non fatti sintatticamente andranno fatti qui nell'analisi semantica. Ad esempio, nella chiamata a funzione vanno anche gestiti i parametri di tipo out del nostro linguaggio MyFun: non solo un parametro formale deve essere *type compatible* con il corrispondente parametro attuale ma entrambi devono

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

avere lo stesso modificatore out o in.

L'output di questa fase è l'AST arricchito con le informazioni di tipo per (quasi) ogni nodo e ed ogni nodo è linkato al proprio type environment.

Per agevolare lo studente nell'implementare l'analizzatore semantico di MyFun, 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, FunOp e BlockOp quando figlio di ProgramOp, o secondo figlio e terzo figlio di IfOp o figlio di WhileOp)
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 anche usare uno stack)

B.
Se il nodo è legato ad un costrutto di *dichiarazione variabile o funzione* (VarDeclOp, ParDeclOp, FunOp) **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. *var x = 10*).

TYPE-CHECK

In questa fase bisogna aggiungere un ***type*** a (quasi) tutti i **nodi** dell'albero (equivalente a dire: dare un tipo ad ogni costrutto del programma) e verificare che le specifiche di tipo del linguaggio siano rispettate.

C.
Se il *nodo* è legato ad un *uso di un identificatore* **allora**
metti in *current_table_ref* il riferimento alla tabella contenuto 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;

// qui è possibile anche memorizzare, nel nodo, il riferimento alla entry nella tabella oltre che il 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"

TYPE SYSTEM

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

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 i due nodi figli hanno raggiunto con successo lo stesso tipo (non error)
THEN l'assegnazione non ha errori di tipo ed il suo tipo finale è NOTYPE
ELSE nodeAssignOp.type = error

costrutti *condizionali*, nodi ifStatOp:
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 la 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

Mancano in questo type system alcune regole di tipo quali ad esempio quelle riguardanti ReadOp, CallFunOp ed altri il cui svolgimento è lasciato allo studente.
Nel file allegato *MyFunTypeSystem.pdf* ci sono alcune delle regole di tipo che utilizzano **regole di inferenza** al posto di regole **IF-THEN-ELSE**.
Le regole di inferenze e le tabelle per il linguaggio MyFun, vanno integrate e consegnate nel documento da allegare alla consegna dell'esercitazione, come sopra indicato.