ESERCITAZIONE 5

Costruire un analizzatore semantico per Toy 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 Toy.

In questa esercitazione, bisogna quindi produrre un compilatore completo che prenda in input un codice Toy e lo compili in un programma C.
Il programma C risultante deve essere compilabile tramite Clang (https://repl.it/languages/c per test online) e deve eseguire correttamente.
Scaricare Clang (ricerca su google tramite la frase "install clang on Mac/Ubuntu/Windows"), e produrre un unico script ***Toy2C*** che metta insieme i due moduli (Toy e Clang tramite comando "clang -pthread -lm -o prog prog.c") e, che, lanciato da linea di comando, compili il vostro programma Toy in un codice eseguibile.

Per testare *Toy2C*, oltre ad utilizzare il programma sviluppato nell'esercizio 4, si sviluppi e compili anche il programma Toy 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: http://a2.pluto.it/a2/a294.htm#almltitle3963)

FACOLTATIVO (Eseguire Toy in un browser web):
Usare Emscripten (https://emscripten.org/docs/getting_started/Tutorial.html) al posto di clang per ottenere una versione web del vostro programma Toy.
A questo punto potete creare lo script **Toy2Web** per compilare un programma Toy in un eseguibile per node.js oppure un 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, Clang (ed eventualmente emscripten) 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 Toy

L'analisi semantica in genere deve svolgere i seguenti due compiti:

Gestione dello scoping: 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 due costrutti[1] che portano alla costruzione di un nuovo scope: il programma ed una dichiarazione di funzione, poiché sono gli unici che prevedono dichiarazioni.
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.

Type checking: utilizzando le tabelle dei simboli, 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** (ovvero le **tabelle dei simboli**) 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 ben tipata e restituisce un intero. **controllare che le callProc siano ben tipate**

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 Toy decidiamo che le funzioni sono sempre dichiarate prima dell'uso per cui una visita dovrebbe bastare.
Si controlli inoltre che esista una ed una sola funzione *main*. Altri controlli possono scaturire da come avete sviluppato la grammatica. Controlli non fatti sintatticamente andranno fatti qui nell'analisi semantica.

L'output di questa fase è l'AST arricchito con le informazioni di tipo per (quasi) ogni nodo e le tabelle dei simboli legate ai nodi.

Per agevolare lo studente nell'implementare il proprio analizzatore semantico di Toy, 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

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

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 la sintassi e la specifica dell'albero sintattico del linguaggio della esercitazione 4)

SCOPING

A.
Se il nodo è legato ad un costrutto di *creazione di nuovo scope* (ProgramOp, ProcOp) **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, magari usando uno stack)

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

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 si potrebbe 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 (ovvero l’intero stack di tabelle correnti) che è dato per scontato e usando i termini IF, THEN.

costrutto *while*, nodo whileOp:
IF il tipo del secondo nodo figlio è *Boolean* AND gli stmt riferiti dal primo e terzo figlio non hanno errori di tipo
THEN il *while* non ha errori di tipo
ELSE nodewhileOp.type = error

costrutto *assegnazione*, nodo AssignOp:
IF i due nodi figli hanno lo stesso tipo
THEN l’assegnazione non ha errori di tipo
ELSE nodeAssignOp.type = error

costrutti *condizionali*, nodi ifStatOp, ElifOp:
IF tipo del primo nodo figlio è *Boolean* e gli altri figli non hanno errori di tipo
THEN non vis ono errori di tipo
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 aritmetici*, 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, CallProcOp ed altri il cui svolgimento è lasciato allo studente.

Nella pagina seguente ci sono alcune definizioni formali elle regole di tipo che utilizzano **regole di inferenza** al posto di regole **IF-THEN-ELSE**. Sono prese dalla specifica del linguagio Pallene ma molte valgono anche per Toy. Ad esempio, quelle della Figura 5 valgono anche per Toy. Nella figura 6 dovremmo modificare quella del while, eliminare quella del for e del local, aggiungere una regola per elif ed altri. Quella del return nel nostro caso non usa la parola chiave “return”.

Le tabelle di compatibilità successive indicano come costruire i tipi per gli operatori unari (sottrazione e not) e binari (addizione, sottrazione, moltiplicazione e divisione). L’operatore unario # non è in Toy e non è da considerare. Si noti che, ad esempio, **optable2(-, integer, integer) = integer** significa che se gli operatori di **-** sono entrambi **integer** allora il tipo risultante è **integer**.

Le regole di inferenze e le tabelle, opportunamente modificate per il linguaggio Toy, vanno consegnate nel documento da allegare alla consegna dell’esercitazione, come sopra indicato.

$$\Gamma \vdash \texttt{nil} : \texttt{nil} \qquad \Gamma \vdash \texttt{true} : \texttt{boolean} \qquad \Gamma \vdash \texttt{false} : \texttt{boolean}$$

$$\Gamma \vdash \mathit{int} : \texttt{integer} \qquad \Gamma \vdash \mathit{flt} : \texttt{float} \qquad \Gamma \vdash \mathit{str} : \texttt{string}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \text{optype1}(op_1, \tau_1) = \tau}{\Gamma \vdash (op_1\ e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \text{optype2}(op_1, \tau_1, \tau_2) = \tau}{\Gamma \vdash (e_1\ op_2\ e) : \tau}$$

$$\frac{\Gamma \vdash f : {\tau_i}^{i \in 1..n} \to \tau \qquad \Gamma \vdash {e_i}^{i \in 1..n} : {\tau_i}^{i \in 1..n}}{\Gamma \vdash f({e_i}^{i \in 1..n}) : \tau}$$

Figure 5: Pallene typing rules for expressions. $\Gamma \vdash e : \tau$ means that expression e has type τ under the environment Γ

$$\Gamma \vdash \texttt{nop} \qquad \frac{\Gamma \vdash \mathit{stmt}_1 \qquad \Gamma \vdash \mathit{stmt}_2}{\Gamma \vdash \mathit{stmt}_1\ \texttt{;}\ \mathit{stmt}_2}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash \mathit{stmt}}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ \mathit{stmt}\ \texttt{end}} \qquad \frac{\Gamma \vdash e_1 : \texttt{integer} \qquad \Gamma \vdash e_2 : \texttt{integer} \qquad \Gamma, x : \texttt{integer} \vdash \mathit{stmt}}{\Gamma \vdash \texttt{for}\ x = e_1, e_2\ \texttt{do}\ \mathit{stmt}\ \texttt{end}}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash \mathit{stmt}_1 \qquad \Gamma \vdash \mathit{stmt}_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{end}} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash \mathit{stmt}}{\Gamma \vdash \texttt{local}\ x : \tau\ \texttt{=}\ e\ \texttt{;}\ \mathit{stmt}} \qquad \frac{(x : \tau) \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x\ \texttt{=}\ e}$$

$$\frac{\Gamma \vdash f : {\tau_i}^{i \in 1..n} \to \texttt{nil} \qquad \Gamma \vdash {e_i}^{i \in 1..n} : {\tau_i}^{i \in 1..n}}{\Gamma \vdash f({e_i}^{i \in 1..n})} \qquad \frac{(\$\mathit{ret} : \tau) \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e}$$

Figure 6: Pallene typing rules for statements. $\Gamma \vdash \mathit{stmt}$ means that the statement stmt is well-typed under the environment Γ. The special entry $\$\mathit{ret}$ in the environment corresponds to the return type for the surrounding function. $\$\mathit{ret}$ is not a valid Pallene identifier, and only appears in these typing rules.

optype1

op_1	operand	result
`-`	`integer`	`integer`
`-`	`float`	`float`
`not`	`boolean`	`boolean`
`#`	`{`τ`}`	`integer`
`#`	`string`	`integer`

optype2

op_2	first operand	second operand	result
`+ - * /`	`integer`	`integer`	`integer`
`+ - * /`	`integer`	`float`	`float`
`+ - * /`	`float`	`integer`	`float`
`+ - * /`	`float`	`float`	`float`
`..`	`string`	`string`	`string`
`and or`	`boolean`	`boolean`	`boolean`
`< == >`	`boolean`	`boolean`	`boolean`
`< == >`	`integer`	`integer`	`boolean`
`< == >`	`integer`	`float`	`boolean`
`< == >`	`float`	`integer`	`boolean`
`< == >`	`float`	`float`	`boolean`
`< == >`	`string`	`string`	`boolean`

Figure 7: Typing relations for primitive operators. Arithmetic operators work on either integers or floating point numbers. The logic operators like `and`, `or` and `not` operate on booleans. Comparison operators work on non-nil primitive types.