ESERCITAZIONE 5

Costruire un analizzatore semantico per YASPL.3 facendo riferimento alle informazioni di cui sotto e legandolo ai due analizzatori già prodotti nell'esercizio 4. Un editore per YASPL.3 è all'indirizzo: http://air.cluelab.di.unisa.it:8080/

Dopo la fase di analisi semantica si sviluppi inoltre un ulteriore visitor del nuovo AST che produca la traduzione **in linguaggio C** di un sorgente YASPL.3.

In questa esercitazione, bisogna quindi produrre un compilatore completo che prenda in input un codice YASPL.3 e lo compili in un programma C.
Il programma C risultante deve essere compilabile tramite Clang (http://releases.llvm.org/download.html) e deve eseguire correttamente.
Si produca quindi un unico script ***yaspl2cc*** che metta insieme i due moduli e, lanciato da linea di comando, compili il vostro programma YASPL.3 in un codice eseguibile.

Per testare *yaspl2cc*, oltre ad utilizzare il programma sviluppato nell'esercizio 4, si sviluppi e compili anche il programma YASPL.3 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:
La seguente istruzione: `clang -S -emit-llvm mioprog.c`
produrrà il codice intermedio LLVM mioprog.ll che potete provare a compilare con Emscripten (http://kripken.github.io/emscripten-site/docs/getting_started/Tutorial.html) per ottenere una versione web del vostro programma YASPL.3.
A questo punto potete creare lo script **yaspl2web** che compili un programma YASPL.3 in una pagina web eseguibile tramite browser.

CONSEGNA
Tutto il codice prodotto e i files di test utilizzati per testare il proprio compilatore vanno sottomessi sulla piattaforma in qualsiasi momento prima dell'orale. Va consegnato 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 (ad esempio, per quanto riguarda l'analisi semantica, le regole di type checking per ReadOp e CallOp)

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 un **programmino 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 YASPL.3

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 solo due costrutti[1] che portano alla costruzione di un nuovo scope: Programma e Def_decl poiché sono gli unici che ammettono dichiarazioni.
Per implementare la most-closely-nested rule le tabelle dei simboli sono oggetto di *push* e *pop* su uno stack (ma mai distrutte) e, per permettere il type checking, restano connesse ai nodi dell'AST pertinenti.

Type checking:utilizzando le tabelle dei simboli, controlla che le variabili siano 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.

Ad esempio, data la dichiarazione int a, b; il costrutto *somma* a + b il type system conterrà la regola

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

oppure, data la dichiarazione di funzione f(out int x, in double y) il costrutto *chiamata a funzione* f(arg1, arg2)

avrà la regola

"IF (f è una funzione che prende un intero come output ed un double come input)
THEN
IF (arg1 è una variabile intera (ma non espressione) e arg2 è una variabile
o un'espressione che restituisce un double)
THEN f(arg1, arg2) ha tipo void
ELSE c'è un type mismatch"

L'analisi semantica è implementata di solito tramite una o più visite depth-first (a seconda della complessità dell'analisi) dell'albero sintattico (AST) legato alla tabella delle stringhe come generato dall'analisi sintattica.
Nel caso di YASPL.3 dovrebbe bastare una sola visita in quanto tutte gli ID vanno dichiarati prima dell'uso.

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 YASPL3, 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. Tutte fanno uso dello stack per far riferimento alla tabella dei simboli corrente.

1 Un costrutto del linguaggio avrà sempre un nodo corrispondente nell'albero sintattico (AST) e molto spesso una produzione corrispondente nella grammatica

(Si legga il seguente testo considerando la sintassi e la specifica dell'albero sintattico del linguaggio della esercitazione 4 (YASPL.3))

SCOPING

A.
Se il nodo è legato ad un costrutto di *creazione di nuovo scope* (ProgramOp, ProcDeclOp) **allora**
se il nodo è visitato per la prima volta **allora**
crea una nuova tabella, legala al nodo corrente e inseriscila al top dello stack (push)

se il nodo è visitato per l'ultima volta (tutti i figli sono stati già visitati) **allora**
eliminala dal top dello stack (pop) [la tabella resta comunque legata al nodo]

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

TYPE-CHECK

In questa fase bisogna aggiungere un ***type*** a 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 al top corrente dello stack

Ripeti
ricerca l'identificatore nella tabella riferita da *current_table_ref* e inserisci il suo riferimento in *temp_entry*.
S**e** l'identificatore non è stato trovato **allora**
current_table_ref = *current_table_ref - 1* // vai giù di 1 nello stack
S**e** *current_table_ref* è andato oltre il fondo dello stack **allora**
restituisci "identificatore non dichiarato"
f**ino 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 (qui bisogna

fare riferimento alle regole del Type System di sotto) **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” [NB: in caso di successo nulla viene inserito nella tabella dei simboli, solo il nodo dell'albero viene aggiornato con il tipo]

TYPE SYSTEM

Questo è un sottoinseme delle regole di tipo definite dal **progettista del linguaggio**.

costrutto *while*, nodo whileOp:
IF il tipo del primo nodo figlio è *Boolean*
THEN *nodo.type* = *void*
ELSE restituisci “type mismatch”

costrutto *istruzione composta*, nodo CompStatOp:
nodo.type = *void*

costrutto *assegnazione*, nodo AssignOp:
IF i tipi dei dei due nodi figli sono tipi compatibili (**definire la propria tabella di compatibilità**)
THEN *nodo.type* = void
ELSE restituisci “type mismatch”

costrutti *condizionali*, nodi ifThenElseOp, ifThenOp:
IF tipo del primo nodo figlio è *Boolean*
THEN *nodo.type* = *void*
ELSE restituisci “type mismatch”

costrutto *operatore relazionale binario*, nodi GtOp, GeOp, etc.:
IF i tipi dei nodi figli primo e secondo sono tipi compatibili con l’operatore (**definire la propria tabella di compatibilità**)
THEN *nodo.type* = *Boolean*
ELSE restituisci “type mismatch”

costrutto *operatore relazionale negato*, nodo NotOp:
IF il tipo del nodo figlio è *Boolean*
THEN *nodo.type* = *Boolean*
ELSE restituisci “type mismatch”

costrutti *operatori aritmetici*, nodi AddOp, MulOp, etc.:
IF i tipi dei dei due nodi figli sono tipi compatibili (**definire la propria tabella di compatibilità**)
THEN *nodo.type* = tipo risultante dalla tabella
ELSE restituisci “type mismatch”

costrutti *operatori booleani*, nodi OrOp, AndOp:
IF il tipo dei due nodi figli è per entrambi *Boolean*
THEN *nodo.type* = *Boolean*
ELSE restituisci “type mismatch”

costrutto operatore *unario minus*, nodo UminusOp:
IF il tipo del nodo figlio è *tipo compatibile con operatore (**definire la propria tabella di compatibilità**)*
THEN *nodo.type* = tipo risultante dalla tabella
ELSE restituisci “type mismatch”

Mancano in questo type system alcune regole di tipo quali ad esempio quelle riguardanti ReadOp e CallOp il cui svolgimento è lasciato allo studente. Vanno anche definite le tabelle di compatibilità che è una matrice che per ogni classe di operatori dice quali sono i tipi compatibili e per questi qual è il tipo risultante.
Es.

Tabella per AddOp

I figlio/II figlio	**Integer**	**Double**	**String**	**Char**	**Bool**
Integer	Integer	Double	error	error	error
Double	Double	Double	error	error	error
String	error	error	error	error	error
Char	error	error	error	error	error
Bool	error	error	error	error	error