LABORATORIO 5

REALIZZAZIONE DI UN LEXER CON ANTLR4 (SimpleExp.g4)
---

1) Realizziamo, con ANTLR4, un lexer che includa i seguenti token
(in ordine di priorità, da quello a priorità più alta a quello a prorità
più bassa):

PLUS associato ai lessemi: la sola stringa '+'
TIMES associato ai lessemi: la sola stringa '*'
LPAR associato ai lessemi: la sola stringa '('
RPAR associato ai lessemi: la sola stringa ')'
NUM associato ai lessemi: la stringa '0' oppure una stringa composta da
    una cifra tra '1' e '9' seguita da
    una qualsiasi quantità (anche nessuna) di cifre tra '0' e '9'

WHITESP associato ai lessemi: una stringa composta da una qualsiasi
    quantità (almeno uno) di caratteri ' ', '\t', '\n' oppure '\r'

ERR associato ai lessemi: stringhe formate da un carattere qualsiasi

dove i token WHITESP ed ERR non devono essere inviati al parser;
ed il token ERR corrisponde ad una condizione di errore (gli errori devono
essere contati e segnalati all'utente).

2) Aggiungiamo al nostro lexer il seguente token "problematico"

COMMENT associato ai lessemi: una stringa che inizia con '/*', contiene
    una qualsiasi quantità (anche nessuna) di caratteri qualsiasi,
    e termina con '*/'

Nel realizzare la specifica per il lexer, fare attenzione alla regola di
maximal-match, che viene applicata per default da ANTLR.

REALIZZAZIONE DI UN PARSER CON ANTLR4 (SimpleExp.g4)

Realizziamo, con ANTLR4, un parser per la grammatica delle espressioni
con operatori + e * vista a lezione.
ANTLR4 (a differenza di versioni precedenti/altri parser top-down) consente

di utilizzare grammatiche ambigue come

E -> E+E | E*E | (E) | n

dichiarando esplicitamente priorità e associatività per i vari operatori (che determinano un unico albero sintattico tra i vari possibili per una stringa).

Ciò rende possibile usare la grammatica in ANTLR senza bisogno di disambiguarla introducendo le variabili T e F come abbiamo fatto a lezione.

Data una stringa corretta, quindi, ANTLR4 produce un unico albero sintattico
(tenendo conto delle regole di priorità/associatività specificate)
senza generare le catene di figli come E-T-F-n.

PRIORITA'
ordine in cui sono scritte le produzioni di una variabile:
la prima produzione ha la priorità più alta
(quindi scriveremo per prima quella dell'operatore "*")

ASSOCIATIVITA'
per ogni produzione (operatore) si ha associatività:
- a destra, se si specifica <assoc=right> prima del corpo della produzione;
- a sinistra, se non si specifica nulla (associatività a sinistra è default)

1) Inseriamo la grammatica sopra in ANTLR4 (file SimpleExp.g4) dando:
priorità al "*" e associatività a sinistra ad entrambi gli operatori.
Affinchè ANTLR4 effettui interamente il parsing del file in input dobbiamo
aggiungere il token speciale EOF alla grammatica.

2) Creiamo un main file "Test.java" (di cui vi viene dato un file iniziale)
che ci consenta di testare il parser/lexer generato automaticamente da ANTLR4
(facendo tasto destro sul file .g4 -> generate ANTLR recognizer)
con il testo nel file "prova.txt" come input.

ESEMPIO:
3+(4+2+7)*5 con eventuali spazi/commenti ed errori lessicali e sintattici

3) Mostriamo come sarebbe banale modificare SimpleExp.g4 in caso volessimo,
invece, la associatività a destra per entrambi gli operatori.

VISITA DEL SYNTAX TREE TRAMITE VISITOR PATTERN: CALCOLO RISULTATO ESPRESSIONE

Possiamo calcolare il risultato di una espressione visitando il relativo albero sintattico, che ANTLR4 genera esplicitamente come un albero di oggetti (facendo preliminarmente tasto destro sul file .g4 -> CONFIGURE ANTLR... -> spunta su "generate parse tree visitor").
Ogni nodo interno dell'albero è di classe "XxxContext" dove "xxx" è il nome di una variabile della grammatica (l'iniziale è resa maiuscola).

In ANTLR4, inoltre, è possibile dare un nome a ciascuna produzione di una variabile (es. "exp") tramite un tag #nome.

1) Consideriamo la grammatica SimpleExp.g4 e diamo un nome a ciascuna produzione della variabile "exp" (usiamo "expProd1" per la prima produzione, "expProd2" per la seconda, ecc...).
La variabile iniziale "prog" ha un'unica produzione, quindi non è necessario dare nomi alle sue produzioni per identificarle.

Come sappiamo ogni nodo interno di un albero sintattico è etichettato con il nome di una variabile, es. "exp", ed ha come figli i simboli del corpo di una produzione di quella variabile, es. la produzione di nome "expProd1".
Nell'albero di oggetti che ANTLR4 produce tale nodo interno sarà di classe effettiva "ExpProd1Context", sottoclasse di "ExpContext" (si veda il file immagine "esempio albero ANTLR").

2) Costruiamo una classe visitor "SimpleCalcSTVisitor.java" (di cui viene dato un file di partenza) per calcolare il risultato delle espressioni. La classe eredita dalla classe, generata automaticamente da ANTLR4, di nome SimpleExpBaseVisitor<E> dove abbiamo scelto Integer come parametro "E" in quanto la visità ritorna un numero intero (il risultato di una espressione).
ESEMPIO: 3+(4+2+7)*5

3) Dotiamo le stampe effettuate durante la visita di indentazione in modo da visualizzare l'albero sintattico generato automaticamente da ANTLR4.

LABORATORIO 6

ESPRESSIONI CON MULTIPLI OPERATORI A STESSO LIVELLO DI PRIORITA' (Exp.g4)

L'utilizzo da parte di ANTLR di grammatiche EBNF (Extended Backus-Naur Form), invece di semplici CFG, consente di utilizzare, all'interno dei corpi delle produzioni, gli operatori delle

espressioni regolari, come: "|" (alternativa), "*" (stella di Kleene), "+" (chiusura positiva) e "?" (presenza opzionale).

Ciò ci consente di estendere la grammatica SimpleExp.g4 aggiungendo a "+" e "*" anche le operazioni "-" e "/" (intero della divisione tra interi), con
- operazioni "*" e "/" allo stesso livello di priorita' (piu' alto)
- operazioni "+" e "-" allo stesso livello di priorita' (piu' basso)
- per entrambi i livelli di priorita' associativita a sinistra

Nei sorgenti di un progetto nuovo chiamato FOOL creiamo un package "exp" in cui collocare i files iniziali forniti dentro la directory "exp" (a parte il file prova.txt, esterno a "exp").

Costruiamo la specifica lessicale/sintattica Exp.g4
- aggiungendo i token DIV e MINUS a quelli di SimpleExp.g4
- modificando la prima e la seconda produzione di SimpleExp.g4 in modo che generino, rispettivamente: l'operazione "*" o "/", e l'operazione "+" o "-".

CALCOLO RISULTATO PER ESPRESSIONI CON MULTIPLI OPERATORI A STESSA PRIORITA'

Nei nodi degli alberi sintattici della grammatica Exp.g4, es. per un nodo di classe "ExpProd1Context", si possono avere due casi: o i suoi figli sono i nodi di una espressione "exp" "*" "exp", o di una espressione "exp" "/" "exp".

1) Costruiamo una classe visitor "CalcSTVisitor.java" modificando il codice di visita in modo da considerare tali due casi.

2) Testiamo l'espressione (15-7+2)*5/10 sia utilizzando la associativita' a sinistra per entrambe le produzioni, sia utilizzando la associativita' a destra.

3) Aggiungiamo la possibilita' di gestire, a livello di sintassi, numeri negativi oltre che positivi ed estendiamo il visitor di conseguenza. Testiamo l'espressione (15-7+2)*-5/10

VISITA SYNTAX TREE TRAMITE VISITOR PATTERN: GENERAZIONE ABSTRACT SYNTAX TREE

All'interno del progetto FOOL creiamo un package "compiler" in cui collocare i files iniziali forniti nella directory "compiler" (a parte il file prova.fool).

Il file FOOL.g4 contiene una prima grammatica incompleta del linguaggio FOOL (Functional Object Oriented Language) che via via estenderemo.
Un primo esempio di codice FOOL e' nel file "esempio.fool". E' un linguaggio funzionale, quindi ogni programma torna un valore senza bisogno di "return"!

Durante la fase di parsing il compilatore genera il syntax tree. Strettamente legata a tale fase vi e' la consecutiva generazione dell'ABSTRACT syntax tree.

1) Usiamo le classi in AST.java per costruire un visitor dei syntax tree delle espressioni di FOOL.g4 aventi gli stessi operatori di quelle di SimpleExp.g4 (chiamato ASTGenerationSTVisitor.g4, di cui viene dato un file iniziale).
Tale visitor deve generare un Abstract Syntax Tree fatto di oggetti "Node".
Ad esempio per il programma FOOL
"(4+2)*-5;"
il visitor deve generare un oggetto fatto come segue:

```
new ProgNode(
  new TimesNode(
    new PlusNode(
      new IntNode(4),
      new IntNode(2),
    ),
    new IntNode(-5),
  )
)
```

2) Testiamo il visitor tramite Test.java (che lo invoca dopo il parsing).

VISITA ABSTRACT SYNTAX TREE (AST) TRAMITE VISITOR PATTERN: STAMPA DELL'AST

Facciamo una semplice visita dell'AST generato (composto da oggetti delle classi in AST.java) in modo da visualizzarlo stampando il nome della classe dei suoi nodi (senza il suffisso Node) in modo indentato.
Per esempio per il programma FOOL considerato prima
"(4+2)*-5;"
il visitor deve stampare:

```
Prog
  Times
    Plus
      Int: 4
      Int: 2
    Int: -5
```

1) Realizziamo una classe PrintASTVisitor (di cui viene dato un file iniziale) che effettui la visita implementando un metodo "visitNode" per ciascuna classe AST.java, che riceva un oggetto di tale classe come argomento (lo lanceremo sulla radice, che e' di classe ProgNode).
Perche' Java da' errore quando proviamo a invocare "visitNode" passando come argomento un nodo figlio?

Mentre, es., in C# esiste un cast "(dynamic)" che consente di determinare il metodo visitNode da invocare a run-time in base al tipo effettivo dell'argomento passato (come dynamic binding ma fatto sull'argomento), cio' non e' possibile in Java: associazione tra invocazione e metodo visitNode invocato fatta a compile-time.

2) In Java dobbiamo usare il metodo classico di implementare il visitor pattern.
a. si crea un metodo visit che riceve un generico Node come parametro n
b. al fine di invocare il metodo visitNode(n) specifico per il tipo effettivo di n:
- si dota ciascuna classe in AST di un metodo accept che invochi visitNode(this)
- si invoca n.accept( ) in modo da utilizzare il dynamic binding di Java su n
NOTA:
- devo aggiungere all'interfaccia Node il metodo accept
- tale metodo deve ricevere l'oggetto PrintASTVisitor (su cui invocare visitNode)

Quando visitavamo i syntax tree utilizzavamo l'implementazione del visitor pattern generata da ANTLR4. Per gli AST abbiamo dovuto implementarlo noi!

3) Testiamo il PrintASTVisitor decommentando le righe in fondo a Test.java

4) Dotiamo il nostro PrintASTVisitor di indentazione automatica usando la stessa tecnica utilizzata per la visita dei syntax tree (es. in ASTGenerationSTVisitor)

5) Al fine di poter realizzare altri tipi di visitor oltre al PrintASTVisitor
utilizziamo un BaseASTVisitor da cui ereditare: creiamo un package di libreria compiler.lib dove mettere le classi generiche Node e BaseASTVisitor

6) BaseASTVisitor contiene: il codice di visit(Node n) e una implementazione
vuota per tutti i metodi visitNode
(il metodo visit delle classi che ereditano da BaseASTVisitor, come PrintASTVisitor, deve invocare il metodo visit con super anziche' direttamente il metodo accept).
Ora le classi in AST devono invocare visitNode(this) su un generico BaseASTVisitor: devo modificare il metodo accept in modo che riceva un generico BaseASTVisitor sia nell'interfaccia Node che nelle classi in AST

7) Usiamo la reflection di Java per fare le stampe, in modo da facilitare l'estendibilita' del linguaggio: decommentiamo funzioni printNode in fondo a PrintASTVisitor e le usiamo per stampare in ogni visita.

VISITA AST CON RITORNO DI UN VALORE (ES. RISULTATO) TRAMITE VISITOR PATTERN

Creiamo un'altra classe per la visita dell'AST. Questa volta vogliamo realizzare un visitor che torni un valore. Come esempio utilizziamo un CalcASTVisitor che calcoli il valore Integer che si ottiene come risultato di un programma FOOL
(per il frammento di FOOL in FOOL.g4, che ad es. non ha chiamate di funzione, e' possibile farlo semplicemente visitando l'AST analogamente a SimpleExp).

1) Modifichiamo la classe BaseASTVisitor introducendo, tramite i generics di Java, un tipo parametrico S da usare come tipo di ritorno per i metodi visit (tale parametro sara' Integer nel CalcASTVisitor e Void nel PrintASTVisitor).
Modifichiamo inoltre il metodo accept nell'interfaccia Node e nelle classi dentro AST.java facendo si' che, anch'esso, torni S.

2) Sistemiamo il PrintASTVisitor in modo che funzioni con la nuova versione parametrica di BaseASTVisitor usando Void
(dichiarare Void come tipo di ritorno mi costringe a mettere "return null;").

3) Testiamo il CalcASTVisitor decommentando le righe in fondo a Test.java

4) Introduciamo una nuova eccezione unchecked "UnimplException" da lanciare
nelle implementazioni di default del metodo visit in BaseASTVisitor.
(importante per accorgerci che abbiamo dimenticato di implementare una visit che viene effettivamente invocata)

LABORATORIO 7

ESTENSIONE FOOL CON DICHIARAZIONE ED USO DI VARIABILI E FUNZIONI

Partiamo da un progetto FOOL avente come sorgenti (nella directory "compiler") i files iniziali forniti nella directory compiler. Il file FOOL.g4 contiene una grammatica estesa del linguaggio FOOL, con anche dichiarazione (con costrutto "let in" di ML) ed uso di variabili e funzioni. Un esempio è nel file esempio.fool

Rispetto alla versione del compilatore che abbiamo sviluppato all'ultimo laboratorio (estesa dall'esercizio per oggi) vi sono le seguenti differenze.

- Stampa dell'AST si può abilitare (ai fini di fare debug) per un qualsiasi ASTVisitor tramite parametro booleano: codice di gestione stampa (campo indent, metodo visit che gestisce l'indentazione e metodi printNode, resi protected) spostati da PrintASTVisitor a BaseASTVisitor. Vecchio metodo visit che invocava accept rinominato in "visitByAcc".
- In ASTGenerationSTVisitor.java il ST visitato si stampa tramite reflection (funzioni usiliare in FOOLlib.java, anche quella per stampa dell'AST) e la stampa si abilita passando "true" al costruttore (modalità debug) come per gli ASTVisitor
- L'interfaccia Node è stata trasformata in classe abstract per includere la gestione del campo "line" che serve per la notifica degli errori (classi in AST ora usano "extends" e sono tutte dotate di tale funzionalità)

Inoltre CalcASTVisitor.java (esempio di AST visitor che torna un valore) non ci serve più.

1) Completiamo l'ASTGenerationSTVisitor in modo da generare l'AST per i nuovi elementi sintattici presenti in FOOL.g4, utilizzando le nuove classi presenti in AST.java. Per ora ci limitiamo a funzioni senza parametri: non consideriamo

- parametri in dichiarazioni
- argomenti in chiamate

ATTENZIONE: gli errori sintattici fanno si' che ANTLR possa fare un match parziale delle produzioni: in questo caso i figli successivi all'ultimo figlio che fa match sono "null", mentre quelli precedenti sono non-"null".
Alcuni token quindi sono sicuramente non-"null": es. il primo token in produzioni che cominciano con un token oppure, ad es., il token PLUS in #plus (la ricorsione a sinistra trasformata internamente in destra lo rende primo token nel ciclo interno in cui sono matchate le produzioni #plus).

2) Testiamo l'ASTGenerationSTVisitor (e stampiamo l'AST generato con il PrintASTVisitor, già esteso con le nuove classi) con il codice in "prova.fool":

```
let
  var y:int = 5+2;
  fun g:bool ()
    let
      fun f:bool ()
        let
          var x:int = -1;
        in g();
    in if y==3 then { f() } else { false };
in
```

```
print (
  if g()
    then { y }
    else { 10 }
);
```

GENERAZIONE DELL'ENRICHED ABSTRACT SYNTAX TREE (EAST) TRAMITE SYMBOL TABLE

Realizziamo la prima fase della semantic analysis vista a lezione: associare usi di identificatori (variabili o funzioni) a dichiarazioni tramite symbol table, usando le regole di scoping statico
- a use of an identifier x matches the declaration in the most closely enclosing scope (such that the declaration precedes the use)
- inner scope identifier x declaration hides x declared in an outer scope

In particolare scegliamo la realizzazione della symbol table come lista (ArrayList) di hashtable (HashMap).

1) Costruiamo una classe SymbolTableASTVisitor (di cui viene dato un file iniziale) che associ usi a dichiarazioni tramite la symbol table
- dando errori in caso di multiple dichiarazioni e identificatori non dichiarati
  (la notifica di errore deve mostrare il numero di linea nel sorgente)
- attaccando alla foglia dell'AST che rappresenta l'uso di un identificatore x
  la symbol table entry (oggetto di classe STentry) che contiene le informazioni
  prese dalla dichiarazione di x (per ora consideriamo solo il nesting level)

(manteniamo la limitazione di considerare solo funzioni senza parametri)

L'effetto della visita è che l'AST si trasforma in un Enriched Abstract Syntax Tree (EAST), dove ad alcuni nodi dell'AST sono attaccate STentry.
E' stato possibile stabilire tale collegamento semantico grazie ai nomi degli identificatori, che da ora in poi non verranno più usati.

2) Testiamo il SymbolTableASTVisitor con lo stesso codice FOOL usato al punto 2) dell'esercizio precedente, facendo piccole modifiche per mostrare gli errori.
(decommentiamo le righe in fondo a Test.java e cancelliamo chiamata a PrintASTVisitor)

VISITA ENRICHED AST (EAST) TRAMITE VISITOR PATTERN: STAMPA DELL'EAST

Facciamo una semplice visita dell'Enriched AST generato in modo da visualizzarlo

stampando in modo indentato, oltre ai suoi nodi (di classe che eredita da Node), anche le sue STentry.

Per farlo dovremo realizzare un visitor che consenta di visitare sia Node che STentry tramite una interfaccia Visitable (contenente il metodo "accept") implementata da entrambi.

1) Aggiungiamo un BaseEASTVisitor che estenda il BaseASTVisitor con un metodo di visita "visitSTentry" con argomento STentry.
(fornisce anche un metodo "printSTentry" per la stampa indentata)

2) Aggiungiamo a STentry il metodo accept, tenendo conto che, sicuramente, riceverà come argomento un BaseEASTVisitor
(codice commentato in fondo a STentry.java)

3) Realizziamo un PrintEASTVisitor che estenda il BaseEASTVisitor: partiamo dal PrintASTVisitor (rinominandolo) e aggiungiamo una implementazione per il metodo di visita "visitSTentry" con argomento STentry.
Invochiamo visit sulle STentry nei nodi a cui sono state attaccate, se presenti.

4) Testiamo il PrintEASTVisitor con lo stesso codice FOOL usato al punto 2) dell'esercizio precedente
(decommentiamo le righe in fondo a Test.java)

LABORATORIO 8

REALIZZAZIONE DI (ENRICHED) AST VISITOR CHE GENERANO ECCEZIONI

In certi casi, per gestire errori rilevati durante una visita, è necessario interrompere la visita lanciando una eccezione (es. perche' è impossibile determinare un valore di ritorno consistente per visitNode in caso di errore).
Ciò sarà necessario per il TypeCheckEASTVisitor che realizzeremo oggi.

Partiamo da un progetto FOOL avente come sorgenti (nella directory "compiler") i files iniziali forniti nella directory compiler. Rispetto alla versione del compilatore che abbiamo sviluppato all'ultimo laboratorio (estesa dall'esercizio per oggi) sono state apportate le seguenti modifiche.

Riguardo la gestione delle eccezioni, è stato aggiunto un nuovo package

"compiler.exc" per le eccezioni, ed è stata realizzata la gestione di visitor
che possono gettare eccezioni come segue.
- BaseASTVisitor:
aggiunto parametro E extends Exception,
aggiunto throws E in TUTTE le visit/visitNode e
aggiunta try-finally in visit che fa indentazione, per ripristinarla comunque
- BaseEASTVisitor: aggiunto parametro E (e passato) e throws E in visitSTentry
- Visitable: ad accept aggiunto binder E, throws E, e parametro E in argomento
- AST: stessa modifica nell'implementazione di accept di tutte le classi
- STentry: stessa modifica più uso parametro E anche nel cast
- PrintEASTVisitor: uso VoidException (unchecked) come parametro in ereditata
(mi consente di chiamare visit senza fare ne' try-catch ne' dich. throws; si
noti che posso sempre overridare un metodo che dichiara throws senza throws)
- SymbolTableASTVisitor: stessa cosa

Le altre modifiche effettuate sono le suguenti.
- classi in AST e metodi di visita riordinati (radici in alto foglie in basso)
- è stata introdotta la classe abstract TypeNode extends Node per i tipi:
TypeNode è ora classe genitore di IntTypeNode e BoolTypeNode, e
i campi type di ParNode/VarNode e retType di FunNode sono ora dei TypeNode
(in ASTGenerationSTVisitor sono stati introdotti cast a TypeNode dopo visite)

ESTENSIONE INFORMAZIONE CONTENUTA IN STentry: TIPI DEGLI IDENTIFICATORI

Estendiamo le informazioni, prese dalla dichiarazione degli identificatori,
contenute nelle symbol table entry (classe STentry) introducendo il tipo:
- un tipo BoolTypeNode oppure IntTypeNode per le variabili o per i parametri
- un tipo funzionale ArrowTypeNode (extends TypeNode) per le funzioni

ArrowTypeNode (classe commentata in fondo a AST.java) contiene le informazioni
corrispondenti alla notazione per i tipi funzionali vista a lezione:
(T1,T2,...,Tn)->T
Cioè il tipo T1,T2,...,Tn dei parametri (nel campo parlist) ed il tipo T di
ritorno della funzione (nel campo ret).

1) Modifichiamo la classe SymbolTableASTVisitor aggiungendo la costruzione e
l'inserimento del tipo nelle STentry (campo type).

2) Modifichiamo la classe PrintEASTVisitor aggiungendo:
- nella visita delle STentry, la stampa del tipo tramite visita del nodo
contenuto nel campo type
- la visita di ArrowTypeNode (da decommentare): richiede una stampa speciale

"marcata" per il tipo di ritorno (oltre ad essere indentato, viene preceduto da "->"), realizzata aggiungendo un parametro a metodo visit di BaseASTVisitor

3) Testiamo il SymbolTableASTVisitor e il PrintEASTVisitor con il codice di esempio in "esempio.fool"

TYPE CHECKING TRAMITE VISITA DELL'ENRICHED ABSTRACT SYNTAX TREE

Realizziamo la seconda fase della semantic analysis vista a lezione: il type checking, che viene effettuato tramite visita dell'enriched abstract syntax tree determinando i tipi delle espressioni (TypeNode) in modo bottom-up.

1) Costruiamo una classe TypeCheckEASTVisitor (di cui viene dato un file iniziale) che realizzi il type checking dei programmi FOOL la cui sintassi è quella di FOOL.g4, a parte la chiamata di funzioni (nodo CallNode)
- la relazione di subtyping è definita tramite il metodo isSubtype di FOOLlib
- consideriamo i booleani essere sottotipo degli interi con l'interpretazione: true vale 1 e false vale 0

In caso il visitor rilevi un errore di tipo deve lanciare una eccezione TypeException contenente il messaggio di errore ed il numero di linea: ciò automaticamente incrementa il contatore "typeErrors" della classe FOOLlib.

2) Testiamo il TypeCheckEASTVisitor con il seguente codice FOOL (in "prova.fool"), facendo piccole modifiche per mostrare gli errori (decommentiamo le righe in fondo a Test.java).

```
let
  var x:int = 5+3;
  fun f:bool (n:int, m:int)
    let
      var x:bool = true;
    in x==(n==m);
in
  print (
    if x==8
      then { false }
      else { 10 }
  );
```

3) Per poter rilevare multipli errori di tipo introduciamo la cattura di TypeException durante la visita, in caso di type checking di dichiarazioni.
La visita di dichiarazioni non torna un oggetto TypeNode (semplicemente torna null) che serva al chiamante: possiamo quindi accettare a questo livello un

errore di tipo avvenuto dentro la dichiarazione senza propagare l'eccezione.
- introduciamo la cattura e stampa di eccezioni quando si visitano dichiarazioni
- facciamo lo stesso in Test per le eccezioni nella main program expression

---
GESTIONE ERRORI PRECEDENTI A TYPE CHECKING: ST ED (ENRICHED) AST INCOMPLETI

Il compilatore deve completare tutte le fasi del front-end anche in presenza di errori collezionando più errori possibili (anche relativi alle fasi precedenti al type checking) in modo che il programmatore possa correggerli insieme.

Il problema però è che:
- errori lessicali/sintattici possono portare a creazione da parte di ANTLR4 di Syntax Tree incompleti (in cui le variabili figlie di un nodo sono "null")
- tali errori (per un effetto a catena) ed errori semantici rilevati via symbol table possono portare alla creazione di EAST che contengono anch'essi figli null
Ciò tipicamente genera null pointer exceptions durante l'esecuzione e impedisce che si continui a collezionare errori per le "parti buone" del programma.

1) Gestiamo i Syntax Tree incompleti tornando null quando si effettua una qualsiasi visit con argomento null nell'ASTGenerationSTVisitor
(questo risolve il problema ma causa la generazione di AST incompleti)

La gestione degli (Enriched) AST incompleti dipende dallo specifico visitor:
- per alcuni visitor è sufficiente lo stesso approccio usato per i Syntax Tree
(es. SymbolTableASTVisitor e PrintEASTVisitor)
- per altri visitor è necessario gettare un'eccezione unchecked IncomplException
(es. TypeCheckEASTVisitor)

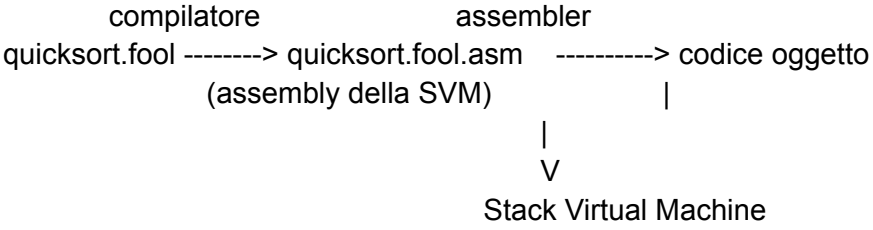
2) Gestiamo i due casi sopra introducendo un parametro booleano "incomplExc" aggiuntivo al BaseASTVisitor e al BaseEASTVisitor che indichi se si vuole che venga, o meno, gettata una IncomplException in caso di albero incompleto.
- in BaseASTVisitor, quando si effettua una qualsiasi visit con argomento null, torniamo null o lanciamo l'eccezione sulla base di tale paramentro
- settiamo appropriatamente tale nuovo parametro in ciascun visitor e in TypeCheckEASTVisitor aggiungiamo la cattura di IncomplException quando si visitano dichiarazioni e in Test.java
(possiamo ora togliere il controllo di STentry a null nel PrintEASTVisitor)

Si noti che la visita dei TypeNode (che ritorna null) è importante, non solo per stamparli in caso di debug, ma anche per controllare che non siano incompleti prima di utilizzarli!

LABORATORIO SVM

REALIZZAZIONE DELLA STACK VIRTUAL MACHINE (SVM) E DEL RELATIVO ASSEMBLATORE

Realizziamo una Stack Virtual Machine (SVM) e il relativo assemblatore.
Esempio: compilazione ed esecuzione file sorgente quicksort.fool



1) Creiamo un progetto FOOL contenente un package "svm" in cui collocare i files iniziali forniti nella directory "svm" (basta copiarla dentro "src" progetto).

2) Creiamo l'assemblatore SVM in SVM.g4 inserendo direttamente in fondo a ogni produzione della grammatica il codice Java da eseguire.

3) Creiamo l'esecutore di codice oggetto SVM in ExecuteVM.java.
TestASM.java consente di assemblare ed eseguire un file ".asm".
Lo testiamo con il file di prova "prova.asm".

PREPARAZIONE PER GENERAZIONE DI CODICE E SUA ESECUZIONE TRAMITE SVM

Aggiungiamo al progetto FOOL contenente il package "svm" della nostra Stack Virtual Machine i files iniziali forniti nella directory "compiler".

Rispetto alla versione del compilatore che abbiamo sviluppato all'ultimo laboratorio (estesa dall'esercizio per oggi) è stato introdotto:
un metodo ausiliario ckvisit(t) da utilizzare quando si leggono tipi t in campi dell'EAST, che lancia la visita su t (per controllarne la completezza e stamparlo in caso di debug) e torna t stesso.

Il codice in fondo a Test.java nella directory "compiler" esegue il CodeGenerationASTVisitor (che realizzeremo, per ora viene fornito uno scheletro)

e poi serializza il contenuto della stringa generata in un file di testo con estensione ".asm".
Si noti che il CodeGenerationASTVisitor viene eseguito solo se il front-end del compilatore non dà errori: non dobbiamo quindi gestire EAST incompleti.

1) Rendiamo "public" i campi "lexicalErrors" del lexer e "code" del parser in SVM.g4 in modo che possano essere acceduti dal Test.java del package compiler.

LABORATORIO 9

CODE GENERATION TRAMITE VISITA DELL'(ENRICHED) ABSTRACT SYNTAX TREE
STEP1: NO DICHIARAZIONE (E CHIAMATA) FUNZIONI

Partiamo da un progetto FOOL avente come sorgenti i files iniziali forniti nella directory "compiler" e nella directory "svm" (contenente i files della nostra Stack Virtual Machine).

Realizziamo l'ultima fase del compilatore: la code generation, che viene effettuato tramite visita dell'(Enriched) Abstract Syntax Tree determinando il codice generato dalle espressioni (String) in modo bottom-up.

Tecnicamente la code generation non richiederà di proseguire la visita dell'AST visitando anche le STentry. Quindi, pur accedendo alle STentry attaccate ai Node, sarà sufficiente un CodeGenerationASTVisitor che genera il codice sotto forma di una stringa Java (si veda esempio in "prova.fool.asm").

1) Inizialmente consideriamo una limitazione al linguaggio FOOL.g4 in cui non si usino ne' dichiarazioni ne' chiamate di funzioni: si ha quindi solo l'ambiente globale, in cui sono dichiarate variabili che vengono utilizzate localmente dalla main program expression (non si hanno nested scopes come a lezione).
Si veda esempio in "prova2.fool".

Sullo stack si ha quindi il solo AR dell'ambiente globale (in cui vengono allocate le variabili dichiarate quando inizia l'esecuzione del programma): utilizziamo come layout per tale AR quello in "layout solo var globali.txt".
Su tale layout sarà basata la generazione di codice.

2) Modifichiamo la classe SymbolTableASTVisitor aggiungendo il calcolo e l'inserimento dell'offset nelle STentry (campo offset) per i VarNode.

3) Modifichiamo la classe PrintEASTVisitor aggiungendo la stampa dell'offset nella visita delle STentry.

4) Costruiamo una classe CodeGenerationASTVisitor (di cui viene dato un file iniziale) che realizzi la code generation dei programmi FOOL la cui sintassi è quella di FOOL.g4, a parte la dichiarazione e chiamata di funzioni (ci si avvale del metodo di FOOLlib nlJoin() che concatena stringhe di codice inserendo un newline "\n" come separatore interno tra stringhe).

5) Testiamo il CodeGenerationASTVisitor con il codice FOOL in "prova2.fool" (decommentiamo le righe in fondo a Test.java che assemblano ed eseguono il codice generato, come già spiegato nella lezione sulla SVM) e visualizziamo il codice generato nel file ".asm".

CODE GENERATION TRAMITE VISITA DELL'(ENRICHED) ABSTRACT SYNTAX TREE
STEP2: DICHIARAZIONE (E CHIAMATA) DI FUNZIONI E NESTED SCOPES

Consideriamo ora il linguaggio completo in FOOL.g4, comprendendo la dichiarazione di funzioni e i nested scopes. Progettiamo il layout degli AR (per ambiente globale e chiamate di funzioni) lavorando sul file "progettiamo nostro layout.txt", dove è stato inizialmente inserito il layout per gli AR delle funzioni considerato delle slides di lezione.

Sul progetto del layout degli AR sarà basata la generazione di codice.

2) Modifichiamo la classe SymbolTableASTVisitor aggiungendo il calcolo e l'inserimento dell'offset nelle STentry (campo offset) per tutte le dichiarazioni: variabili, funzioni e parametri.

La code generation di usi di identificatori (usi di variabili IdNode o chiamate di funzioni CallNode) richiede di conoscere la differenza di nesting level tra l'uso (IdNode/CallNode) e la relativa dichiarazione (campo "nl" della STentry). Dobbiamo quindi dotare anche IdNode e CallNode di un campo "nl".

3) Modifichiamo la classe SymbolTableASTVisitor aggiungendo l'inserimento del nesting level nel campo "nl" di IdNode e CallNode.

4) Modifichiamo la classe PrintEASTVisitor aggiungendo la stampa del campo "nl" nella visita di IdNode e CallNode.

Commentando l'esecuzione del CodeGenerationASTVisitor in Test.java possiamo

visualizzare l'Enriched AST per l'esempio in "prova.fool": negli IdNode e CallNode si vede la differenza di nesting level con la relativa dichiarazione.

5) Modifichiamo la classe CodeGenerationASTVisitor realizzando la code generation dei programmi FOOL la cui sintassi è quella di FOOL.g4, a parte il codice generato per il corpo delle funzioni.