Progetto di Linguaggi e Compilatori – Parte 3
A.A. 2021-22

Gruppo 21-3

Attenzione: una volta letto il presente testo è *obbligatorio* consegnare alla scadenza quanto elaborato, indipendentemente dal fatto che lo si consideri adeguato o meno.

Si ricordi che questa è una prova d'esame, pertanto **si deve produrre un contributo originale, sviluppato in autonomia**. Si può utilizzare solo il materiale didattico fornito durante il corso ed eventualmente il materiale didattico fornito pubblicamente da altri docenti in altri corsi, compresi libri di testo (a meno che non sia in contraddizione con il materiale del corso). In particolare **NON È PERMESSO** utilizzare (anche solo parti di) soluzioni di progetti realizzati da altri studenti, di qualunque corso di studi, in qualunque anno, di qualsiasi università.

Entro il termine stabilito si *deve* inviare via email dentro un unico file compresso, il cui nome sia "`ProgettoLC parte3 Gruppo 3`", tutti (e soli) i sorgenti sviluppati, in modo che sia possibile verificare il funzionamento di quanto realizzato. In tale file compresso *non* devono comparire file oggetto e/o binari o qualsiasi altro file che viene generato a partire dai sorgenti. Sempre in detto file va inoltre inclusa una relazione in formato PDF dal nome "`ProgettoLC parte3 Gruppo 3 Relazione`". La relazione può contenere immagini passate a scanner di grafici o figure fatte a mano, il che **non** include fotografie digitali. Evitare scansioni di scritti a matita e assicurarsi comunque che si abbia un buon contrasto. Volendo si può concordare, al momento del ritiro di questo testo, la consegna *entro la scadenza* di documentazione cartacea (per non dover includere lo scan nella relazione PDF).

- Si richiede una descrizione dettagliata di tutte le tecniche **non**-standard impiegate (le tecniche standard imparate a lezione non vanno descritte).
- Si richiede una descrizione delle assunzioni fatte riguardo alla specifica, sia relativamente a scelte non previste espressamente dalla specifica stessa, che a scelte in contrasto a quanto previsto (con relative motivazioni).

  Non è assolutamente utile perdere tempo per includere nella relazione il testo dei vari esercizi o un suo riassunto o una qualsiasi rielaborazione, incluso descrizioni del problema da risolvere. Ci si deve concentrare solo sulla descrizione della soluzione e delle eventuali variazioni rispetto a quanto richiesto.
- Si richiede una descrizione sintetica generale della soluzione realizzata.
- Si richiede di commentare (molto) sinteticamente ma adeguatamente il codice prodotto.
- Si richiede di fornire opportuno Makefile (compliant rispetto allo standard GNU make) per
  - poter generare automaticamente dai sorgenti i files necessari all'esecuzione (lanciando `make`);
  - far automaticamente una demo (lanciando `make demo`).
- La soluzione *deve* essere implementata in Haskell, utilizzando Happy/Alex. Si suggerisce di utilizzare BNFC per costruire un primo prototipo iniziale da raffinare poi manualmente, ma non è obbligatorio (si includa anche il sorgente .cf in caso).

Esercizio

Si consideri un semplice linguaggio imperativo, con scoping statico e tipi espliciti, costruito a *partire* dalla stessa *sintassi concreta* del linguaggio **Pascal**.

In tale linguaggio (che non è Pascal) si possono dichiarare funzioni/procedure, oltre che variabili, all'interno di qualsiasi blocco.

Si hanno le procedure predefinite `writeInt`, `writeReal`, `writeChar`, `writeString`, `readInt`, `readReal`, `readChar`, `readString`. Le `read`... hanno argomento per riferimento.

Si hanno le operazioni aritmetiche, booleane e relazionali standard ed i tipi ammessi siano

- i tipi base: interi, booleani, real, caratteri, stringhe, e
- i tipi composti: array (di tipi qualsiasi) e puntatori (a tipi qualsiasi),

con i costruttori e/o le operazioni di selezione relativi. Il tipo 'interi' deve essere compatibile con il tipo 'real' mentre viceversa il tipo 'real' **non** deve essere compatibile con il tipo 'interi'.

Il tipo 'caratteri' deve essere compatibile con il tipo 'interi'. Viceversa il tipo 'interi' non deve essere compatibile con il tipo 'caratteri'.

Il tipo dei predicati di uguaglianza e disuguaglianza deve essere tale da poter confrontare fra loro **solo** i tipi base "atomici", cioè booleani, caratteri, interi e real ($\forall \tau \in \{bool, char, int, real\}. \tau \times \tau \to bool$).

Gli altri predicati di confronto devono poter confrontare fra loro solo tipi aritmetici (cioè i tipi che son compatibili con real).

Non è necessario avere tipi di dato definiti dall'utente.

Il linguaggio deve implementare—con la sintassi concreta di Pascal qualora sia prevista—le modalità di passaggio dei parametri value, reference e value-result, con i relativi vincoli di semantica statica.

Il linguaggio deve supportare, oltre agli assegnamenti semplici, anche gli assegnamenti che coinvolgono un'operazione, almeno per le operazioni `+`, `-` e `*`. Si utilizzi la semantica canonica per tali operazioni (non altre varianti). Per la sintassi concreta di questi costrutti (qualora non fosse già espressamente prevista in qualche forma in Pascal) si scelga una forma consona al resto della sintassi concreta.

Il codice TAC generato deve introdurre espliciti controlli a run-time sull'ammissibilità degli indici degli array (cioè controllare che il valore di un indice sia compreso nei limiti previsti dalla dimensione dell'array). In caso di errore il codice TAC deve chiamare una procedura predefinita (di cui ci si inventerà il nome e che si suppone blocchi l'esecuzione).

Il linguaggio deve prevedere—con la sintassi concreta di Pascal qualora sia prevista—gli usuali comandi per il controllo di sequenza (condizionali semplici, le due forme canoniche di iterazione *indeterminata*) ed almeno un costrutto di iterazione *determinata*.

Dentro il corpo dei cicli *indeterminati* si devono ammettere le istruzioni `break` e `continue`, con la sintassi concreta di Pascal, qualora sia prevista. Fuori dal corpo dei cicli (indeterminati o determinati) tali istruzioni non devono essere ammesse (dentro il corpo dei cicli *determinati* a piacere).

Per detto linguaggio (non necessariamente in ordine sequenziale):

- Si progetti (e implementi) una opportuna sintassi astratta e si implementi la corrispondente funzione di serializzazione che trasformi un albero di sintassi astratta in sintassi concreta *legale* con un minimo di ordine, ad esempio andando a capo in corrispondenza dei blocchi).
- Si progetti un type-system (definendo le regole di tipo rispetto alla sintassi astratta generata dal parser, eventualmente semplificandone la rappresentazione senza però snaturarne il contenuto semantico).
  - Si rappresentino espressamente in forma grafica le relazioni di compatibilità fra tutti i tipi semplici previsti. Si specifichino formalmente le relazioni fra tutti i tipi composti previsti.
  - Si specifichi espressamente quale sia la visibilità effettiva di qualsiasi tipo di dichiarazione (per esempio "solo dal punto di dichiarazione fino alla fine" oppure "in tutto il blocco" oppure altro).
  - Per i costrutti di iterazione determinata si specifichi se la variabile di iterazione si deve considerare come una (implicita) dichiarazione locale del corpo o meno (e si scelga una forma di sintassi concreta consona o opportuni vincoli sul tipo).
- Si implementi un lexer con Alex, un parser con Happy. Si suggerisce di utilizzare BNFC per costruire un primo prototipo iniziale da raffinare poi manualmente, ma non è obbligatorio (si includa anche il sorgente .cf in caso).
- Si implementi il type-checker e *tutti* gli altri opportuni controlli di semantica statica (ad esempio il rilevamento di utilizzo di r-expr illegali nel passaggio dei parametri). Si cerchi di fornire messaggi di errore che aiutino il più possibile a capire in cosa consiste l'errore, quali entità coinvolge e dove queste si collochino nel sorgente.
- Dopo aver definito un opportuno data-type con cui codificare il three-address code (non una stringa di char!), si implementi il generatore di three-address code per il linguaggio considerato (non Pascal), tenendo presente che:
  - gli assegnamenti devono valutare l-value prima di r-value;
  - l'ordine delle espressioni e della valutazione degli argomenti si può scegliere a piacere (motivandolo);
  - le espressioni booleane nelle guardie devono essere valutate con short-cut. In altri contesti si può scegliere a piacere (motivandolo).

Si predispongano dei test case significativi per una funzione che, dato un nome di file, esegua il parsing del contenuto, l'analisi di semantica statica, il pretty-print del sorgente ed (infine) generazione e pretty-print del three-address code. Nel pretty-print del three-address code si serializzino gli identificatori che vengono dal programma aggiungendo l'informazione relativa al punto di dichiarazione nel sorgente originale (ad esempio `t37 = x_12 + 5` se `x` è stata dichiarata alla linea 12).

Importante: si tenga presente che del linguaggio Pascal si devono utilizzare *solo* le scelte di sintassi concreta per i costrutti (canonici) previsti dal testo precedente. Tutto il resto, sia la sintassi concreta di costrutti non richiesti sia la semantica di funzionamento, è irrilevante ai fini della soluzione e probabilmente anche in contrasto con quanto si richiede di fare. Argomentazioni del tipo "ma in Pascal queste cose non esistono" o "in Pascal ci sono questi costrutti che funzionano così" non saranno accettate.