Progetto di Linguaggi e Compilatori – Parte 5
A.A. 2019-20

Gruppo 2

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

Entro il termine stabilito si *deve* inviare via email dentro un unico file compresso, il cui nome sia "`ProgettoLC parte5 Gruppo 2`", 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 parte5 Gruppo 2 Relazione`". La relazione può contenere immagini passate a scanner di grafici o figure fatte a mano.

Si richiede:

- una descrizione dettagliata di tutte le tecniche **non**-standard impiegate (le tecniche standard imparate a lezione non vanno descritte).
- 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.
- una descrizione sintetica generale della soluzione realizzata.
- di commentare (molto) sinteticamente ma adeguatamente il codice prodotto.
- 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`).
- di implementare la soluzione in Haskell, usando Happy/Alex (o BNFC).

Esercizio

Si consideri un semplice linguaggio imperativo, con scoping statico e tipi espliciti, costruito a *partire* dalla stessa *sintassi concreta* del linguaggio **Lua** (home page http://www.lua.org/).

In tale linguaggio, contrariamente a Lua, i parametri formali di procedure/funzioni vengono dichiarati, oltre che con i tipi, anche con modalità esplicite (tranne che per la modalità "di default" di passaggio per valore che può essere opzionale).

Si possono dichiarare funzioni/procedure, oltre che variabili, all'interno di qualsiasi blocco.

Si hanno le procedure/funzioni predefinite `writeInt`, `writeFloat`, `writeChar`, `writeString`, `readInt`, `readFloat`, `readChar`, `readString`.

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

- i tipi base: interi, booleani, float, 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 'float' mentre viceversa il tipo 'float' **non** deve essere compatibile con il tipo 'interi'.

Il tipo 'caratteri' *non* deve essere compatibile con il tipo 'stringhe' (ne tantomeno il viceversa ovviamente).

Il tipo dei predicati di uguaglianza e disuguaglianza deve essere tale da poter confrontare fra loro **tutti** i tipi base ('interi' con 'interi', 'stringhe' con 'stringhe', ...) mentre gli altri predicati di confronto devono poter confrontare fra loro solo tipi aritmetici (quindi non sia valido per 'stringhe' con 'stringhe', etc.).

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

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

Il linguaggio deve ammettere le istruzioni `break` e `continue` (dentro il corpo dei cicli indeterminati), con la sintassi concreta di Lua, qualora sia prevista.

Al linguaggio si aggiunga inoltre un costrutto *if-then-else* per la categoria sintattica delle espressioni (oltre che all'*if-then-else* dei comandi). Per la sintassi concreta di questo costrutto (qualora non fosse già espressamente prevista in qualche forma in Lua) si scelga una forma consona al resto della sintassi concreta.

Il linguaggio deve prevedere—con la sintassi concreta di Lua 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*.

Per detto linguaggio (non necessariamente in ordine sequenziale):

- Si progetti una opportuna sintassi astratta.
- 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 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 ed il corrispondente pretty printer (che serializzi un albero di sintassi astratta in sintassi concreta *legale* con un minimo di ordine, ad esempio andando a capo in corrispondenza dei blocchi). Si suggerisce di utilizzare BNFC per costruire un primo prototipo iniziale da raffinare poi manualmente, ma non è obbligatorio.
- 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 Lua), 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 Lua 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 Lua queste cose non esistono" o "in Lua ci sono questi costrutti che funzionano così" non saranno accettate.