

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 parte3parz 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 parte3parz 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).
- 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 **Chapel** (home page <http://chapel.cray.com>).

In tale linguaggio (che non è Chapel) le procedure vengono dichiarate come funzioni di tipo **void**. Si possono dichiarare funzioni(procedure), oltre che variabili, all’interno di qualsiasi blocco.

Si hanno le funzioni predefinite **writeInt**, **writeReal**, **writeChar**, **writeString**, **readInt**, **readReal**, **readChar**, **readString**.

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.

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

Il linguaggio deve prevedere—con la sintassi concreta di Chapel qualora sia prevista—gli usuali comandi per il controllo di sequenza (condizionali semplici, iterazione *indeterminata*).

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 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 includa anche il sorgente .cf in caso).

Progetto di Linguaggi e Compilatori – *PARZIALE* Parte 3

A.A. 2020-21

- Si implementi il type-checker. 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 Chapel), 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 Chapel 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 Chapel queste cose non esistono” o “in Chapel ci sono questi costrutti che funzionano così” non saranno accettate.