Progetto di Linguaggi di programmazione 2015-2016 Parte Quinta: Traduzione da LKC al linguaggio per la macchina virtuale `SECD`

G. Filè

16 dicembre 2015

1 Introduzione

La quinta parte del progetto consiste nel completare un compilatore che traduce i programmi LKC in corrispondenti programmi nel linguaggio per la macchina virtuale `SECD`, che chiameremo semplicemente, linguaggio `SECD`. Il documento è organizzato nel modo seguente. Nella Sezione 2 verrà illustrata la macchina virtuale `SECD` e le sue istruzioni. Nella Sezione 3 discuteremo la traduzione dei programmi LKC in programmi `SECD`. Questa traduzione fornisce la semantica operazionale del linguaggio LKC e in definitiva anche del LispKit. La Sezione finale contiene la consegna da eseguire.

2 La macchina `SECD` e il suo linguaggio

La macchina `SECD` consiste di quattro componenti `<S,E,C,D>` il cui ruolo viene spiegato nel seguito:

- La pila `S` serve, durante l'esecuzione dei programmi `SECD`, per contenere i risultati parziali e finali dei calcoli; `S` è una pila e le operazioni che vengono eseguite trovano i loro argomenti sulla cima della pila, li *consumano* lasciando il risultato al loro posto sulla cima della pila.

- L'ambiente `E` modella la RAM che contiene gli R-valori delle variabili del programma in esecuzione. Durante l'esecuzione, nuove variabili (col loro R-valore) vengono introdotte dai costrutti `LETC, LETRECC` con i binders e da invocazioni di funzioni con i parametri attuali che danno valore ai corrispondenti parametri formali della funzione invocata. Ciascuno di questi costrutti costituisce un nuovo blocco e questi blocchi corrispondono ai record d'attivazione (RA) che costituiscono l'ambiente dinamico `E`. Per modellare la sequenza di RA, `E` è una lista di liste di valori. Ogni lista di valori modella un RA e contiene gli R-valori delle variabili dichiarate nel corrispondente blocco (binders e parametri formali). La

lista in cima alla pila `E` è il blocco corrente e durante la sua permanenza in cima a `E` vengono eseguite le istruzioni del corrispondente blocco. Ogni R-valore contenuto in `E` è identificato da una coppia di interi `(n1,n2)` in cui `n1` individua la `n1`-esima lista di `E` (`n1`=0 identifica la lista in cima a `E`, `n1`=1 è la seconda lista e così via) e `n2` identifica l'`n2`-esimo valore della `n1`-esima lista.

Questi indirizzi vengono usati dal comando SECD `Ld` (che corrisponde al comando assembler `LOAD`) e che ha il seguente compito: `Ld(n1,n2)` sposta l'R-valore di indirizzo `(n1,n2)` da `E` alla cima della pila `S`. Quindi gli indirizzi `(n1,n2)` fanno parte del codice SECD prodotto dalla compilazione di un programma LKC. Capire come il compilatore produce questi indirizzi è un punto importante dell'intera traduzione. Infatti il compilatore non possiede l'ambiente `E`, che si sviluppa solo durante l'esecuzione, ma il compilatore costruisce durante la compilazione stessa una copia di `E` costituita non dagli R-valori delle variabili (noti solo a tempo di esecuzione), ma semplicemente dai nomi delle variabili stesse. Quindi il compilatore usa e costruisce una lista di liste di stringhe (appunto i nomi delle variabili), che sarà ovunque denotato con `n`, in cui il significato delle liste che la compongono è esattamente lo stesso che in `E`. In sostanza durante la compilazione si riesce a costruire un modello `n` dell'ambiente `E` sufficientemente informativo da garantire che gli indirizzi `(n1,n2)`, calcolati a tempo di compilazione, durante l'esecuzione delle istruzioni `LD(n1,n2)`, trovino gli R-valori desiderati in `E`. Per sottolineare la stretta relazione che esiste tra `E` ed `n`, ma anche l'importante differenza tra di essi, li chiameremo rispettivamente, **ambiente dinamico** e **ambiente statico**.

- Il controllo `C` è il programma `SECD` che si intende eseguire sulla macchina `SECD`;
- Il dump `D` è un deposito in cui viene salvato lo stato della macchina `SECD` per esempio quando viene invocata una funzione oppure quando viene eseguito il comando `SECD` condizionale, vedi la lista che segue.

Descriviamo ora il linguaggio `SECD`. Si deve tenere a mente che il significato dei diversi comandi di questo calcolo è di far evolvere la macchina `SECD`. Il compilatore produce programmi `SECD` e quindi per realizzarlo correttamente è indispensabile capire bene cosa fanno le operazioni di questo linguaggio. Come per i programmi LKC, definiamo nel seguito un tipo Haskell che serve a rappresentare in modo semplice i programmi `SECD`. Questo tipo possiede un costruttore per ogni operazione `SECD` e nel seguito illustreremo quale trasformazione dello stato della macchina `SECD` comporta l'esecuzione di ciascuna di queste operazioni .

```
data Secdexpr = Add | Sub |  Mult | Div | Rem | Eq | Leq |
                Car | Cdr | Cons | Atom | Join | Rtn | Stop | Push |
                Ap | Rap | Ld (Integer, Integer) |
                Ldc LKC|
                Sel [Secdexpr] [Secdexpr]|
                Ldf [Secdexpr]
```

```
deriving(Show, Eq)
```

Segue la spiegazione dei diversi costruttori del tipo `secdexpr` e delle operazioni SECD che essi rappresentano:

- I costruttori `Add, Sub, Mul, Div, Rem, Eq, Leq, Car, Cdr, Cons, Atom, Join, Rtn, Ap, Stop, Rap, Push` rappresentano le operazioni 0-arie della `SECD`. Tutte queste operazioni a parte le ultime 6 (`Join, Rtn, Ap, Stop, Rap, Push`) hanno un significato immediato e che corrisponde alle analoghe operazioni LKC. Può forse sorprendere il fatto che non abbiano operandi (mentre in LKC li hanno). Il motivo è che nella macchina `SECD`, gli operandi di questi operatori vengono messi in cima allo stack `S` (attraverso operazioni `Ld` o `Ldc`, illustrate nei punti successivi) prima che venga eseguito l'operatore che li usa. Il risultato dell'operazione viene poi inserito in cima allo stack `S` della `SECD` al posto degli operandi appena usati.

  Per quanto riguarda gli ultimi 6 operatori, il `Join` viene spiegato più sotto assieme al costruttore `Sel`, mentre `Rtn` ed `Ap` verranno descritti subito dopo la presente lista di punti. L'operazione di `Stop` è ovvia. Le operazioni `Rap` e `Push` servono per il caso di definizioni locali ricorsive `LETRECC` che per il momento viene trascurato. Quindi non verranno spiegate in questo documento.

- `Ld(Integer, Integer)`: è il comando di caricamento sulla cima della pila `S` della `SECD` del valore che compare nella posizione `(n1,n2)` dell'ambiente `E`. Precisamente, visto che `E` è una lista di liste di valori, `Ld(n1,n2)` carica sulla cima dello stack `S` l'n2-esimo valore dell'n1-esima lista di `E` (contando a partire dalla cima).

- `Ldc of LKC`: è il comando che carica sulla pila `S` un valore costante senza valutarla. I valori che verranno caricati sono `NUM(int), BOO(B), STRI(string)` e `NIL`. Insomma le costanti del tipo LKC che il compilatore lascia inalterate e che la macchina `SECD` ricopierà sullo store `S` e valuterà al momento opportuno.

- `Sel [secdexpr] [secdexpr]`: è il ben noto comando condizionale che, a seconda del valore di verità in cima alla pila `S`, esegue le istruzioni del suo primo parametro (caso then) oppure del secondo (ramo else). Nel seguito, per semplicità, i valori booleani sono indicati con T e F per true e false. Più precisamente se la situazione della `SECD` è la seguente:

  ```
  (T S)  E  ((Sel C_T C_F) C) D --> S  E  C_T  (C D)
  ```

  Quindi il programmma `C` che segue `Sel` viene memorizzato sul dump `D` e viene eseguito `C_T` visto che sullo stack `S` c'è `T` che rappresenta `true`. Se invece in cima alla pila ci fosse `F` verrebbe eseguito `C_F`. Sia `C_T` che `C_F` devono avere come ultima istruzione la `Join` la cui esecuzione produce il seguente effetto:

```
S  E  (Join) (C D) --> S  E  C  D
```

cioè si restaura il controllo `C` salvato sul dump dall'esecuzione di `Sel`.

- `Ldf [secdexpr]`: è il comando che viene eseguito in corrispondenza della definizione di una funzione e che serve a costruire in cima alla pila `S` la chiusura della funzione che è definita. Per capire la prossima spiegazione è necessario ricordare che la chiusura di una funzione è una coppia che consiste del corpo della funzione (tradotto in lista di comandi `SECD`) e dell'ambiente `E` al momento dell'esecuzione dell'`Ldf` (cioè al momento della definizione della funzione). Notare che l'ambiente in questo momento è, in generale, diverso da quelli che saranno correnti nei momenti in cui la funzione verrà invocata (che, in generale, saranno tutti diversi). Quindi ricordare l'ambiente al momento della definizione della funzione permette di realizzare lo **static binding** per le eventuali variabili globali nel corpo della funzione. L'effetto dell'esecuzione di `Ldf` sulla `SECD` è il seguente:

  ```
  S  E  (Ldf(C') C)  D --> ((C' E) S) E  C  D
  ```

  Come si vede viene inserita sullo stack la chiusura `(C' E)`.

È arrivato il momento di spiegare le istruzioni `SECD` `Ap` e `Rtn` (`Ap` sta per APply=invoca) e `Rtn` (per ReTurN). L'istruzione `Ap` ha il seguente effetto:

```
((C' E')A  S) E  (Ap C)  D -->  NIL  (A E')  C'  ((S E C) D)
```

Questa istruzione viene sempre eseguita quando in cima alla pila `S` c'è una chiusura `(C' E')` ed immediatamente sotto alla chiusura c'è la lista `A` dei valori dei parametri attuali dell'invocazione. L'effetto di `Ap` è di caricare il corpo `C'` della funzione sul controllo (per eseguirla) e di costruire l'ambiente in cui fare questa esecuzione nel modo seguente. L'ambiente in cui il corpo della funzione viene eseguito è `A E'`. In questo modo quando il corpo viene eseguito, saranno disponibili i parametri attuali `A` dell'invocazione e, in `E'`, gli R-valori delle variabili attive al momento della definizione della funzione. Si osservi che questo garantisce la realizzazione dello **static binding**. Il fatto che la lista `A` dei valori dei parametri attuali vada messa in cima all'ambiente in cui eseguire il corpo della funzione, va confrontato con quello che il compilatore fa dei parametri formali di una funzione (LAMBDA) quando compila il suo corpo: il compilatore mette la lista dei parametri formali in cima all'ambiente statico `n` (che simula a tempo di compilazione l'ambiente dinamico `E` della SECD). La traduzione della definizione di una funzione (LAMBDA), operata dal compilatore, è illustrata nella Sezione 3.

Oltre alle operazioni appena descritte, l'esecuzione di `AP` salva sul dump `D` la tripla `(S E C)` in modo da ripristinarla quando l'esecuzione della funzione finisce e qui possiamo già dire che il corpo della funzione deve sempre terminare con l'istruzione `Rtn` che si occupa di fare questo ripristino. L'effetto di `Rtn` è il seguente:

```
xS'  E'  (Rtn)  (S E C D) --> xS  E  C  D
```

Si osservi che l'ambiente `E'` viene eliminato e che il valore x, che è il risultato calcolato dalla funzione, viene messo a disposizione del calcolo che segue inserendolo in cima allo stack ripristinato dal dump: `xS`. Il calcolo continua con il programma `C`.

3 Compilazione da LKC → a SECD

Chiameremo COMP il compilatore da LKC a `SECD` che va completato in quest'ultima parte del progetto. COMP consiste di una funzione ricorsiva con il seguente tipo:

```
COMP:: LKC -> [[LKC]] -> [Secdexpr]->[Secdexpr]
```

se chiamiamo `e, n` e `c` i suoi 3 parametri formali, essi hanno il seguente significato:

- `e:LKC` è il programma LKC che deve essere tradotto in linguaggio `SECD` e il risultato di questa traduzione è il valore restituito da COMP;
- `n:[[LKC]]` è l'ambiente statico (vedi Sezione 2), cioè la lista di liste di variabli (rappresentate come del data type LKC) che sono presenti quando `e` viene compilato. In pratica `n` rappresenta la stack dei record d'attivazione del programma eseguito prima di arrivare ad eseguire `e`. Quindi, all'inizio della compilazione `n` dovrebbe essere `[]`, insomma vuoto.
- il terzo parametro `c:[secdexpr]` contiene ad ogni istante della compilazione il codice `SECD` prodotto fino a quell'istante. Non è indispensabile, ma rende la definizione di COMP più agevole. Da quanto detto segue che inizialmente `c=[]`. Procedendo con la compilazione, alla lista di istruzioni `SECD c` vengono aggiunte nuove istruzioni in testa alla lista stessa. Questo può sembrare strano per chi è abituato ai programmi imperativi in cui la compilazione inizia creando il codice che corrisponde all'inizio del programma e procede appendendo le successive istruzioni alla destra. I programmi LKC sono valori di tipo `LKC` per cui hanno una forma che ci permette di accedere immediatamente alle diverse componenti di ciascuna espressione in modo da compilarle nell'ordine giusto che è quello tale che quando il codice `SECD` verrà eseguito da sinistra verso destra produrrà il risultato atteso. Per esempio, se il programma `LKC` da compilare consiste dell'invocazione di una funzione con certi parametri attuali, verrà per prima cosa prodotto e aggiunto in testa a `c` il codice relativo all'invocazione della funzione e successivamente si aggiungeranno, sempre in testa a `c`, i programmi `SECD` che calcolano e mettono sullo stack `S` i parametri attuali. In questo modo, quando verrà eseguito il codice `SECD` contenuto in `c`, verranno innanzitutto calcolati e messi su `S` i parametri attuali e poi verrà eseguita l'invocazione della funzione.

La funzione COMP(e,[],[]) da il significato (operazionale) di ogni espressione `e:LKC` (e quindi dei programmi LispKit). Infatti il significato di `e` è la sequenza di configurazioni della macchina `SECD` che sono attraversate eseguendo il programma COMP(e,[],[]) a partire dalla configurazione iniziale della macchina virtuale `SECD`, cioè `(S=[], E=[], C=COMP(e,[],[]), D=[])`.

La funzione COMP è descritta nel file `compilatore.hs` che trovate sul moodle del corso assieme a questo file. In questa funzione mancano i casi corrispondenti ai costrutti `LETC` e `LETRECC` che costituiscono la Parte 5 del progetto. Le parti della funzione COMP che sono date, dovrebbero essere per la maggior parte comprensibili, comunque di seguito trovate qualche spiegazione.

- `VAR ''x''`: la funzione `location` calcola l'indirizzo `(n1,n2)` della variabile `x` in `n` e l'istruzione `Ld(n1,n2)`, nella macchina SECD, ha il compito di caricare sullo stack `S` l'R-valore della variabile `x`.

- `NUM(n)`: indica che `n` è una costante intera e quindi viene tradotto in `Ldc(NUM(n))`. Lo stesso succede con le altre costanti del tipo LKC: `BOO(b),STRI(s)` e `NIL` che vengono tradotte rispettivamente in `Ldc(BOO(b)),Ldc(STRI(s))` e `Ldc(NIL)`.

- `ADD(x,y)`: il codice `SECD` che viene prodotto è tale che quando viene eseguito, carica sullo stack `S` il valore di `y` (è l'esecuzione di `(COMP y n _)` che avrà questo effetto), poi carica in modo analogo il valore di `x` e poi ne fa la somma eseguendo l'operazione `SECD, Add`. Si osservi che in questo modo, al momento di eseguire `Add`, il primo operando `x` è in cima alla pila `S`, ed immediatamente sotto di lui c'è `y`. Tutte le altre operazioni, binarie o unarie funzionano nello stesso modo.

- `IF(a,b,c)`: il codice `SECD` prodotto è tale che quando viene eseguito carica sullo stack `S` il valore di `a` che è la condizione del condizionale, dopo viene eseguito `Sel(thenp, elsep)` e questo comporta l'esecuzione di `thenp` o di `elsep` a seconda del valore di verità di `a` che è in cima ad `S`. L'esecuzione di `Sel` ha anche l'effetto di spostare del dump `D` il contenuto della componente `C` della macchina `SECD`. La successiva esecuzione dell'istruzione di `Join` che conclude entrambi `thenp` e `elsep`, riporta il codice memorizzato in `D` nella componente `C`. Questo è un modo di ottenere che, sia che venga eseguito il ramo then, sia che venga eseguito l'else, una volta terminata questa esecuzione, il calcolo continui dall'istruzione immediatamente successiva al condizionale (cioè dal programma `C` salvato su `D`). Un altro modo di fare la stessa cosa sarebbe quello di ricopiare il codice `C` dopo `thenp` e anche dopo `elsep` al posto di mettere l'istruzione di `Join`, ma questo ovviamente duplicherebbe inutilmente il codice.

- `LAMBDA(parform, body)`: COMP produce codice `SECD` corrispondente al corpo `body` usando un ambiente statico `n` a cui viene aggiunta all'inizio la lista dei parametri formali `parform` della funzione. Il motivo di aggiungere ad `n` i parametri formali è che ogni volta che questa funzione verrà invocata (durante l'esecuzione del programma SECD), i suoi parametri attuali saranno preventivamente messi in cima all'ambiente dinamico `E` (cf. istruzione `Ap` della Sezione 2) ed occuperanno quindi la stessa posizione che i loro nomi occupano nell'ambiente statico quando il corpo della funzione è compilato. Il programma SECD che viene prodotto da COMP per il corpo della funzione termina sempre con l'istruzione `Rtn` che esegue il ritorno, cioè toglie dal dump `D` la prima tripla `(S E C)`

e la usa per inizializzare, rispettivamente, la pila, l'ambiente dinamico e il controllo della macchina SECD, cf. istruzione Rtn nella Sezione 2. Intuitivamente, in questo modo si ripristina la situazione della macchina prima dell'invocazione e l'effetto dell'invocazione è costituito unicamente dal suo risultato che si troverà in cima alla pila S. Si ricordi che la tripla (S E C) viene messa in cima al dump D dall'istruzione Ap che esegue l'invocazione della funzione.

Si assuma che la compilazione di body (corpo della funzione dichiarata con la lambda) sia B. Allora, B sarà parametro di un'istruzione Ldf la quale, quando verrà eseguita, caricherà B sullo stack S, costruendo, allo stesso tempo, la chiusura (B,E), dove E è l'ambiente dinamico al momento dell'esecuzione di Ldf, cioè al momento della definizione della funzione stessa. Questo ambiente E serve a realizzare lo scope statico. Per capire cosa succede di questa chiusura, quando la funzione viene invocata, si guardi l'istruzione Ap descritta nella Sezione 2.

E' utile inquadrare le operazioni appena descritte in una visione più ampia. La definizione di una funzione può essere un binder di un LETC o LETRECC oppure un parametro attuale dell'invocazione di una funzione. In tutti i casi viene costruita la chiusura della funzione ed essa viene messa sullo stack S, come parte della lista dei binders/parametri utilizzati dall'esecuzione del corpo del LETC, LETRECC o della funzione invocata. Al momento di questa esecuzione i binders/parametri (alcuni dei quali possono essere chiusure) andranno sull'ambiente dinamico E, da cui potranno venire prelevati con opportune operazioni Ld.

Si confrontino queste osservazioni con i successivi punti che illustrano la traduzione di CALL e di LETC.

- CALL(paratt,nome): il codice SECD prodotto è tale che quando viene eseguito costruisce sullo stack S la lista dei parametri attuali, la traduzione del nome sarà semplicemente Ld(n1,n2) dove (n1,n2) è la posizione nell'ambiente dinamico E della chiusura che è il valore della funzione invocata (si osservi che l'indirizzo (n1,n2) è calcolato dal compilatore usando la sua copia statica n dell'ambiente dinamico E). Questa istruzione Ld(n1,n2) carica la chiusura sullo stack S. Su S, i parametri sono immediatamente sotto la chiusura e a questo punto l'istruzione Ap, che il compilatore produce subito dopo Ld(n1,n2), produce il salvataggio su D della situazione corrente della macchina e l'esecuzione della funzione nell'ambiente dinamico contenuto nella sua chiusura a cui si aggiunge la lista dei parametri attuali. È importante osservare che la lista di parametri attuali viene inserita in cima a quest'ambiente esattamente come i nomi dei parametri formali erano stati messi in cima ad n nella compilazione del corpo di una funzione (vedi il caso LAMBDA più sopra). La preparazione dei parametri formali viene fatta tramite la funzione complist che è data. Questa funzione, che svolge lo stesso compito anche nei casi LETC e LETRECC, merita di essere studiata con attenzione.

- LETC(a,b) e LETRECC(a,b): la traduzione di questi casi non è specificata e in questo consiste la quinta parte del progetto. Per aiutarvi a realizzare questa traduzione osserviamo che il LETC è in qualche modo l'unione del caso CALL

e del caso `LAMBDA`. Infatti una `LETC` introduce binders della forma `x=exp`, in cui le variabili locali `x`, per il corpo del `LETC`, sono del tutto simili ai parametri formali del caso `LAMBDA`. Quindi il corpo del `LETC` va compilato mettendo queste variabili locali in cima all'ambiente statico `n` usato dal compilatore. D'altra parte i valori delle variabili locali, cioè i valori dei corrispondenti `exp`, hanno lo stesso ruolo dei parametri attuali di una `CALL` e quindi, in corrispondenza di questi valori, va prodotto codice `SECD` che costruisca una lista di questi valori sullo stack `S`. Sopra questa lista dovrà venire inserita la chiusura del corpo del `LETC` e in questa situazione, una `Ap` farà eseguire il corpo del `LETC` nell'ambiente dinamico corretto (con i valori delle variabili locali disponibili ed al posto giusto, cioè in cima all'ambiente dinamico).

Per quanto riguarda la traduzione, il `LETRECC` è molto simile al `LETC`. Una differenza è che la traduzione deve usare `Rap` al posto di `Ap` (R sta per Recursive, naturalmente). Oltre a questo, si deve fare attenzione che la traduzione dei binders sia effettuata in un ambiente statico che contenga le variabili locali dei binders del `LETRECC`, cioè le parti sinistre dei binders. Questo corrisponde al fatto che quando le funzioni ricorsive, definite nelle parti destre dei binders, verranno eseguite, questo deve avvenire in un ambiente dinamico che contiene le chiusure delle funzioni stesse. Solo così infatti si possono eseguire le invocazioni ricorsive. Questo effetto viene ottenuto costruendo delle chiusure circolari per le funzioni ricorsive definite nei binders del `LETRECC`, cioè chiusure che nella seconda componente (ambiente di definizione della funzione) contengono la chiusura stessa. Queste chiusure circolari sono realizzate dal comando `Rap` che usa, a questo scopo, le funzioni `LazyE` e `LazyClo`, contenute nel programma dell'interprete SECD. La circolarità è realizzata in modo banale visto che Haskell è lazy. In un `LETRECC` si possono inserire anche binders non ricorsivi, purché la variabile a sinistra di questi binder non venga usata in qualche parte destra di binder. Insomma non ci può essere circolarità tra binders non ricorsivi. Cioè non è consentito avere una coppia di binders come x=y and y=x (ovviamente), ma neppure come x=2 and y=x. Quest'ultimo sarebbe gestibile, ma complicherebbe inutilmente l'interprete. Quindi l'interprete che vi è dato non è capace di gestire questi casi correttamente. Per poter gestire binder non ricorsivi (che devono essere noncircolari), è necessario che la compilazione del `LETRECC` introduca nel codice SECD che produce la costante PUSH, descritta nel punto seguente.

- `Push` serve a a fare in modo che l'interprete SECD inserisca un segnaposto [OGA] (vedi il data type Valore dell'interprete) nel suo ambiente dinamico. L'ambiente di ogni chiusura ricorsiva dovrà sempre iniziare con [OGA], al cui posto verrà sostituito dall'interprete l'ambiente circolare che serve per le funzioni ricorsive. Il segnaposto serve a garantire che gli indirizzi calcolati dal compilatore per le variabili non ricorsive che appaiono nelle parti destre dei binders trovino i valori giusti durante l'esecuzione. Il record d'attivazione segnaposto [OGA], che viene inserito nell'ambiente dinamico delle chiusure ricorsive, corrisponde alla lista delle parti sinistre dei binders aggiunta all'ambiente statico quando si compilano le parti destre dei binders. La successiva esecuzione

di `Rap` sostituisce il segnaposto [OGA] con l'ambiente circolare richiesto dai binder ricorsivi.

4 Consegna

Nel file `compilatore-incompleto.hs` viene fornito il codice Haskell di un compilatore da LKC a codice `SECD`. Le parti riguardanti i casi `LETC` e `LETRECC` mancano. Si richiede di realizzarli.

Oltre a `compilatore-incompleto.hs` viene fornito il file `interprete.hs` che contiene la funzione Haskell `interprete S E C D`, i cui parametri rappresentano le 4 componenti della macchina `SECD` e che esegue i programmi `SECD` in modo fedele al significato di ciascuna operazione descritto nella Sezione 2. Potete usarlo per eseguire i programmi `SECD` prodotti dal compilatore. L'interprete è completo. Non è richiesta alcuna aggiunta. È richiesta però la comprensione delle istruzioni della macchina `SECD`.