Progetto di programmazione avanzata e parallela

Parte 2 di 2 - Linguaggio Python

Aggiornato il 2024-01-01

Lo scopo di questo progetto è quello di estendere il codice prodotto durante l'undicesima esercitazione (19 dicembre 2023) permettendo di supportare le seguenti funzionalità:

Operatori di confronto

Deve essere fornito il supporto per gli operatori `>`, `>=`, `=`, `!=`, `<` e `<=`. Questi operatori devono ritornare valori booleani.

Definizione di variabili

Per rendere il linguaggio più simile a un reale linguaggio di programmazione è necessario poter definire variabili, in particolare:

- `var alloc` dove `var` è una variabile. Rende disponibile la variabile per le parti successive del codice. Questo può essere fatto modificando l'ambiente (`env`). Il valore di default della variabile deve essere zero. La valutazione non ritorna alcun valore.
- `n var valloc` dove `var` è una variabile e `n` il risultato della valutazione di una espressione che ritorna un valore intero. Rende disponibile un array di `n` elementi (di default tutti zero)
- `expr x setq` imposta il valore della variabile `x` al risultato dell'espressione `expr`, ritorna il nuovo valore di `x`.
- `expr n x setv` imposta il valore dell'`n`-esima posizione dell'array indicato dalla variabile `x` al valore ritornato da `expr`. Si noti che anche `n` può essere una espressione. Ritorna il nuovo valore di `x` in posizione `n` (gli array sono indicizzati da zero).

Sequenze

Deve essere possibile eseguire in sequenza più operazioni:

- `expr1 expr2 prog2`, `expr1 expr2 expr3 prog3`e `expr1 expr2 expr3 expr4 prog4` valutano le due, tre o quattro espressioni precedenti ritornando il valore di `expr1`

Condizionali e iterazioni

Deve esserci il supporto per

- `if-no if-yes cond if` Se la valutazione di `cond` ritorna un valore vero allora viene valutata `if-yes` altrimenti `if-no`. Viene ritornato il valore del ramo dell'if che è stato valutato.
- `expr cond while`, valuta `cond` e, se è vera, valuta `expr` finché `cond` non diventa falsa.

- `expr end start i for`. Valuta `expr` più volte con il valore di `i` (una variabile) da `start` a `end - 1` con incrementi di 1

Subroutine

È possibile definire subroutine (che non prendono argomenti) associando del codice a una variabile che può poi venire eseguito facendo una chiamata tramite `call`:

- `expr f defsub`. Non valuta `expr` ma l'associa alla variabile `f`, che potrà poi essere chiamata tramite `call`
- `f call`. Valuta l'espressione associata a `f` (definita tramite `defsub`).

Funzionalità aggiuntive

- `expr print`. Valuta `expr` e stampa il risultato. Ritorna il valore di `expr`.
- `nop`. Non esegue nessuna operazione.

Esempi di codice

```
x 1 + x setq x 10 > while x alloc prog2
```

- Definisce la variabile `x`
- Finché `10 > x`:
  - `x = x + 1`

```
v print i i * i v setv prog2 10 0 i for 10 v valloc prog2
```

- Alloca un vettore `v` di 10 elementi
- Per `i` che va da `0` (incluso) a `10` (escluso):
  - Imposta `v[i] = i * i`
  - Stampa `v`

```
x print f call x alloc x 4 + x setq f defsub prog4
```

- Definisci la subroutine `f` come:
  - `x = x + 4`
- Definisci la variabile `x`
- Chiama la subroutine `f`
- Stampa la variabile `x`

```
nop
i print
i x % 0 = if
1000 2 i for
783 x setq
x alloc
prog3
```

- Definisci la variabile `x`
- Imposta il valore di `x` a `783`
- Per `i` che va da `2` (incluso) a `1000` (escluso):

    - Se `x % i == 0`:
        * Stampa `i`
        * Altrimenti non fare nulla

Il risultato è un programma che trova i divisori di 783.

```
nop
x print
prime if
nop
0 0 != prime setq
i x % 0 = if
1 x - 2 i for
0 0 = prime setq
prime alloc
prog4
100 2 x for
```

- Per `x` da `2` (incluso) a `100` (escluso):
    - Dichiara la variabile `prime`
    - Imposta il valore di `prime` a *true* (i.e., `0 == 0`)
    - Per `i` da `2` (incluso) a `i - 1` (escluso):
        * Se `i` divide `x` allora:
            · Imposta `prime` a *false* (i.e., `0 != 0`)
            · Altrimenti non fare nulla
    - Se `prime` è *true*:
        * Stampa `x`
        * Altrimenti non fare nulla

Ovvero il programma stampa tutti i numeri primi inferiori a 100.

Requisiti

- In caso di errore il codice deve generare delle eccezioni coerenti con il tipo di condizione che ha provocato l'errore. È possibile definire delle eccezioni apposite o usare quelle predefinite (si sconsiglia sollevare eccezioni troppo generiche, come `Exception`).
- Ogni funzione rilevante (i.e., non di 2-3 righe) deve essere adeguatamente commentata spiegando che compito svolge, che input richiede e che output genera.
- Ogni file .py deve contenere all'inizio come commento il proprio nome, cognome e numero di matricola.

Consegna

Le date di consegna dipendono dalle date degli appelli orali:

- Appello del 17/01/2024:
    - Apertura consegna: 03/01/2024

  - Chiusura consegna: 10/01/2024 (ore 23:59)
- Appello del 14/02/2024:
  - Apertura consegna: 31/01/2024
  - Chiusura consegna: 07/02/2024 (ore 23:59)

Per appelli successivi verranno comunicate le date di consegna. Non inviate al di fuori delle date di consegna.

Istruzioni per la consegna:

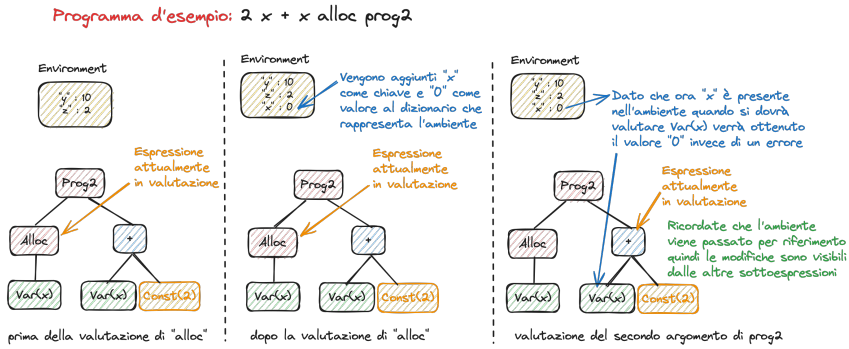
- Preparare il progetto e salvarlo come un file `.zip` o `tar.gz` contenente tutti i file *sorgenti* necessari. **NON** devono essere allegati i file binari. Può essere allegato un file `README.txt` o `README.md` con istruzioni e note.
- Inviare una mail a `lmanzoni@units.it` con il seguente oggetto:
  `[Consegna Progetto Programmazione Avanzata e Parallela] Nome Cognome Matricola`
  usando la vostra email istituzionale. Dovete chiaramente allegare il progetto, per il testo della mail non vi sono vincoli.

Note importanti

- Questo progetto è relativo solamente alla parte in Python del corso. Per sostenere la parte di progetto del corso è necessario consegnare sia la parte in C (parte 1) che la parte in Python (parte 2).
- Sebbene appaia ovvio, quando viene chiesto di inserire Nome, Cognome e Matricola, dovete inserire il vostro nome, cognome e numero di matricola a non, letteralmente, il testo "Nome", "Cognome" e "Matricola".
- Il progetto rimane valido anche per gli appelli successivi, non è necessario inviarlo nuovamente.
- Ogni consegna successiva annulla quella precedente, si sconsiglia però di inviare decine di versioni durante il periodo di consegna.
- Devono essere sempre consegnate entrambe le parti, **NON** solo una delle due.

Addendum

- Dato che `dispatch` contiene le informazioni su quali siano le *keywords* del linguaggio che stiamo definendo, il dizionario andrà modificato aggiungendo le associazioni per le operazioni che vengono man mano aggiunte (e.g., `for`, `alloc`, etc.).
- Dato una istruzione `op` con n argomenti, l'ordine degli argomenti da considerare è dall'n-esimo come il più a sinistra e il primo quello più a destra. Per esempio per `i x %` l'operazione di modulo ha come primo argomento `x` e come secondo argomenti `i`, dovendo quindi essere interpretato come `x % i`.
- Si ricorda che istruzioni come `alloc`, `valloc`, `setq` e `setv` modificano l'ambiente passato come argomento a `evalutate`, come illustrato in figura.



Changelog

- 2024-01-01. Specifica dell'ordine degli argomenti aggiunta in **Addendum** e aggiunta di chiarimenti sul funzionamento dell'ambiente da passare come argomento a `evaluate`.
- 2023-12-29. Correzione dell'indentazionde del secondo esempio. Aggiunte informazioni sul trattamento di `dispatch` in **Addendum**.