Progetto di Linguaggi e Programmazione Orientata agli Oggetti

a.a. 2020/2021

Modificare il codice dell'interprete del linguaggio L proposto come soluzione dell'ultimo laboratorio di LPO per implementare la sua estensione L^{++}.

Sintassi delle espressioni: L^{++} permette l'uso dell'operatore binario infisso `!=` di confronto, i literal di tipo *range*, che rappresentano sequenze di numeri interi, e l'operatore unario prefisso `bounds` che restituisce gli estremi di una sequenza di tipo *range*. Nella grammatica il non-terminale `Exp` contiene le seguenti produzioni aggiuntive:

```
Exp ::= ... | Exp != Exp | [Exp:Exp] | bounds Exp
```

Nota bene:

- `bounds` è una nuova keyword.
- Le produzioni specificate sopra vanno disambiguate in modo che l'operatore `!=` abbia lo stesso livello di precedenza di `==` ed associ, quindi, a sinistra.

  La seguente tabella riassuntiva specifica le precedenze tra tutti gli operatori binari infissi, in ordine crescente di precedenza (`&&` è l'operatore a precedenza più bassa).

operatori
&&
== !=
+
*

Sintassi degli statement: il linguaggio L^{++} include anche lo statement `for` che permette di iterare l'esecuzione del suo blocco sugli elementi di una sequenza di tipo *range*.

```
Stmt ::= ... | for IDENT in Exp Block
```

Nota bene: `for` e `in` sono nuove keyword.

Semantica statica

La semantica statica è specificata dal programma OCaml nel file `semantica-statica.ml`.
I literal `[Exp:Exp]` hanno tipo *range*.
La semantica statica dell'operatore di confronto `!=` è la stessa di quella dell'operatore `==`: i due operandi devono avere lo stesso tipo statico e il risultato ha tipo *bool*.
L'operatore `bounds` è definito solo se l'operando ha tipo *range*; il risultato ha tipo prodotto $int*int$ (ossia, il tipo di una coppia di interi).
Lo statement `for IDENT in Exp Block` è corretto staticamente rispetto all'ambiente *env* solo se

- l'espressione `Exp` ha tipo statico *range* rispetto all'ambiente *env*;
- il blocco `Block` è corretto staticamente rispetto al nuovo ambiente ottenuto da *env* aggiungendo i seguenti due nuovi livelli di scope annidati:
  - livello 1 (più esterno): contiene solo la variabile `IDENT` di tipo *int*;
  - livello 2 (più interno): non contiene variabili.

Per esempio, il programma

```
var x=true;
for x in [1:3]{
   var x=x!=0;
   print x
};
print x
```

è staticamente corretto e la sua esecuzione stampa `true` tre volte.

Semantica dinamica

La semantica dinamica è specificata dal programma OCaml contenuto nel file `semantica-dinamica.ml`.
La semantica dell'operatore di confronto `!=` è complementare a quella di `==`.
La semantica di un literal `[`v_1`:`v_2`]` di tipo *range* è la seguente: Se v_1 o v_2 non è un numero intero, allora viene sollevata un'eccezione di tipo `EvaluatorException`; altrimenti vengono distinti i seguenti casi:

- se $v_1 < v_2$ allora il literal rappresenta la sequenza crescente di interi $v_1, v_1 + 1, \ldots, v_2 - 1$ (v_1 incluso, v_2 escluso);
- se $v_1 > v_2$ allora il literal rappresenta la sequenza decrescente di interi $v_1, v_1 - 1, \ldots, v_2 + 1$ (v_1 incluso, v_2 escluso);
- se $v_1 = v_2$ allora il literal rappresenta la sequenza vuota.

Due valori di tipo *range* sono uguali solo se rappresentano la stessa sequenza; per esempio, `[1:1]` e `[0:0]` sono uguali, sebbene abbiano estremi diversi, poiché entrambi rappresentano la sequenza vuota.
Suggerimento importante: in Java conviene implementare i valori di tipo *range* con una classe che implementa l'interfaccia `Iterable`.
La semantica di `bounds` v è così definita:
se il valore v non ha tipo *range* allora viene sollevata un'eccezione di tipo `EvaluatorException`; se il valore v ha tipo *range* e corrisponde a `[`a`:`b`]` allora viene restituita la coppia `<<`a`, `b`>>` dei suoi estremi (a inclusivo, b esclusivo).
La semantica dello statement `for IDENT in Exp Block` rispetto all'ambiente *env* è così definita:

1. l'espressione `Exp` viene valutata rispetto all'ambiente *env*, se il risultato non è un valore r di tipo *range* allora viene sollevata un'eccezione di tipo `EvaluatorException`;
2. un nuovo ambiente *env2* viene creato a partire da *env* aggiungendo uno scope annidato dove la sola variabile `IDENT` è dichiarata con valore iniziale 0 (tale valore è ininfluente ai fini della semantica);
3. il blocco `Block` viene eseguito per tutti i valori della sequenza rappresentata dal valore r di tipo *range* assegnati alla variabile `IDENT` dichiarata in *env2*; ogni esecuzione del blocco avviene rispetto a un ambiente ottenuto a partire da *env2* aggiungendo uno scope annidato inizialmente vuoto;
4. una volta terminata l'esecuzione del `for` viene rimosso lo scope annidato che dichiara `IDENT`.

Suggerimento importante: in Java conviene implementare l'esecuzione dello statement `for` mediante un iteratore (preferibilmente con l'*enhanced for*) evitando la ricorsione.

Per i valori di tipo *range* lo statement `print` stampa il corrispondente literal senza spazi bianchi. Per esempio, il programma `print [1+2 : 2*3]` stampa `[3:6]`

Interfaccia utente

Il progetto implementa la seguente interfaccia utente da linea di comando.

- Il programma da eseguire viene letto dal file di testo *filename* con l'opzione `-i` *filename* oppure dallo standard input se nessuna opzione `-i` viene specificata.
- L'output del programma in esecuzione viene stampato sul file di testo *filename* con l'opzione `-o` *filename* oppure sullo standard output se nessuna opzione `-o` viene specificata.
- L'opzione `-ntc` (abbreviazione di no-type-checking) permette di disabilitare il controllo di semantica statica del type-checker .

Esempi di uso corretto dell'interfaccia, assumendo che la classe principale del progetto sia `interpreter.Main`:

- legge il programma dallo standard input, stampa l'output sullo standard output:
  `$ java interpreter.Main`
- legge il programma dallo standard input, stampa l'output sullo standard output, disabilita il type-checking:
  `$ java interpreter.Main -ntc`

- legge il programma dallo standard input, stampa l'output sul file `output.txt`:
  `$ java interpreter.Main -o output.txt`
- legge il programma dal file `input.txt`, stampa l'output sullo standard output:
  `$ java interpreter.Main -i input.txt`
- legge il programma dal file `input.txt`, stampa l'output sul file `output.txt`:
  `$ java interpreter.Main -o output.txt -i input.txt`
- legge il programma dal file `input.txt`, stampa l'output sul file `output.txt`, disabilita il type-checking:
  `$ java interpreter.Main -o output.txt -ntc -i input.txt`

Le opzioni possono essere specificate in qualsiasi ordine e una stessa opzione può essere ripetuta più volte; in questo caso l'opzione considerata sarà solo l'ultima. Ogni opzione `-i` o `-o` deve essere necessariamente seguita dal corrispondente nome del file.
L'esecuzione del progetto segue il seguente flusso di esecuzione:

1. Il programma in input viene analizzato sintatticamente; in caso di errore sintattico, viene stampato sullo standard error il messaggio associato alla corrispondente eccezione sollevata e il programma termina. Se non vengono sollevate eccezioni, allora l'esecuzione passa al punto 2 se **non** è stata specificata l'opzione `-ntc`, altrimenti passa al punto 3.
2. Viene eseguito il type-checking; in caso di errore statico, viene stampato sullo standard error il messaggio associato alla corrispondente eccezione sollevata e il programma termina. Se non vengono sollevate eccezioni l'esecuzione passa al punto 3.
3. Il programma viene eseguito; in caso di errore dinamico, viene stampato sullo standard error il messaggio associato alla corrispondente eccezione sollevata e il programma termina.

Qualsiasi altro tipo di eccezione dovrà essere catturata e gestita stampando su standard error la traccia delle chiamate sullo stack e terminando l'esecuzione; ogni file aperto dovrà comunque essere chiuso correttamente prima che il programma termini.
L'output dell'interprete **non** deve contenere stampe di debug, ma solo quelle prodotte dalla corretta esecuzione del programma interpretato.