Progetto di Linguaggi e Programmazione Orientata agli Oggetti

a.a. 2017/2018

Modificare l'interprete Java del linguaggio L di riferimento dei laboratori 10 e 11, per implementare il seguente linguaggio esteso L^{++}.

Sintassi

La sintassi di L^{++} è un'estensione di quella di L: tutte le definizioni lessicali e sintattiche di L rimangono valide per L^{++}.

Categorie lessicali: L^{++} è arricchito con due nuove categorie lessicali:

- literal di tipo boolean rappresentati dalle due keyword **`false`** e **`true`**;
- literal interi di tipo binario: iniziano con la cifra `0`, seguita dalla lettera `b` o `B` e terminano con una sequenza **non vuota** di cifre binarie. L'interpretazione è quella convenzionale: per esempio, il literal `0b0111` denota il numero 7.

Sintassi delle espressioni: oltre ai literal specificati sopra, vengono aggiunti gli operatori binari infissi **`&&`** e **`==`** e gli operatori unari prefissi **`!`**, **`opt`**, **`empty`**, **`def`** e **`get`**, secondo le seguenti nuove produzioni (da aggiungere alla grammatica di L presentata nel testo del laboratorio 10):

```
Exp ::= ... | Exp && Exp | Exp == Exp | ! Exp | opt Exp | empty Exp | def Exp | get Exp
```

Nota importante: **`opt`**, **`empty`**, **`def`** e **`get`** sono nuove keyword. Per trattare correttamente il riconoscimento dei lessemi **`=`** e **`==`** tramite un'espressione regolare, elencare prima **`==`**; quindi l'espressione regolare di forma `... ==|= ...` è quella corretta.
Gli operatori **`&&`** e **`==`** associano entrambi a sinistra. Tutti gli operatori unari prefissi hanno la precedenza sugli operatori binari infissi.
Le precedenze tra gli operatori binari infissi sono specificate dalla seguente tabella, in ordine crescente di precedenza (**`&&`** è l'operatore a precedenza più bassa).

operatori
&&
==
::
+
*

Sintassi degli statement: vengono aggiunti due nuovi tipi di statement, secondo le seguenti produzioni (da aggiungere alla grammatica di L presentata nel testo del laboratorio 10):

```
Stmt ::= ... | if (Exp) {StmtSeq} (else {StmtSeq})? | do {StmtSeq} while (Exp)
```

Nota importante: **`if`**, **`else`**, **`do`** e **`while`** sono nuove keyword. Secondo le convenzioni dello stile EBNF, nello statement **`if`** la parte introdotta dalla keyword **`else`** è opzionale.

Semantica statica

La semantica statica è specificata formalmente dal programma OCaml contenuto nel file `spec.ml`.
La sintassi delle espressioni di tipo viene estesa con la costante `bool` e il costruttore unario `opt`. Il primo rappresenta i valori di tipo boolean, il secondo i valori *opzionali* restituiti dagli operatori del linguaggio **`opt`** and **`empty`**; per esempio, le espressioni **`opt`** `1`, **`empty opt true`**, e **`opt`** `[1]` hanno, rispettivamente, tipo `int opt`, `bool opt` e `int list opt`.
La semantica statica per le espressioni boolean è quella convenzionale: **`false`** e **`true`** hanno tipo `bool`, gli operatori `!` e `&&` richiedono che gli operandi siano di tipo `bool` e restituiscono valori di tipo `bool`.
L'operatore == richiede che entrambi gli operandi siano dello stesso tipo e restituisce un valore di tipo `bool`.
La semantica statica per gli operatori che agiscono su valori opzionali è così definita:

`opt` richiede un operando di qualsiasi tipo t e restituisce un valore di tipo t `opt`;

`empty` richiede un operando di tipo t `opt` e restituisce un valore di tipo t `opt`;

`def` richiede un operando di tipo t `opt` e restituisce un valore di tipo `bool`;

`get` richiede un operando di tipo t `opt` e restituisce un valore di tipo t.

Per gli statement `if` `(`e`){...}(else {...})?` e `do{...}while(`e`)`, la guardia e deve avere tipo `bool`.

Nota importante: gli statement dentro i blocchi `{...}` sono a un livello di scope più annidato, come accade per lo statement `for`. Per esempio, il programma

```
var x=1;
if(x==1){var x=[1];x=2::x};
print x==1
```

è staticamente corretto.

Semantica dinamica

La semantica dinamica è specificata formalmente dal programma OCaml contenuto nel file `spec.ml`.
I due literal `false` e `true` si valutano nei corrispondenti valori di verità, `!` è l'operatore di negazione, `&&` è la congiunzione (and) logica, con la strategia di valutazione convenzionale (adottata anche in Java): l'operando di destra viene valutato solo se l'operando di sinistra è stato valutato nel valore *true*.
L'operatore `==` restituisce il valore *true* se, e solo se, i due operandi si valutano in valori uguali. L'uguaglianza tra valori interi e boolean è quella convenzionale.
Due valori lista sono uguali se hanno la stessa lunghezza e contengono elementi uguali a parità di posizione. Per esempio, l'espressione `[1,2]==[1,2] && !([1]==[1,2]) && !([1,2]==[2,1])` si valuta in *true*.
Due valori opzionali sono uguali se sono entrambi indefiniti o se sono entrambi definiti e contengono valori uguali. Per esempio, l'espressione

```
opt 1==opt 1 && empty opt 1==empty opt 0 && !(opt 1==opt 0)
    && !(opt 1==empty opt 1) && !(empty opt 1==opt 1)
```

si valuta in *true*.
L'operatore `opt` restituisce il valore opzionale che contiene il valore ottenuto dalla valutazione dell'operando. Un valore opzionale può contenere un altro valore opzionale.
L'operatore `empty` restituisce il valore opzionale vuoto (o indefinito), ossia quello che non contiene alcun valore. Se il valore dell'operando non è opzionale, la valutazione solleva un errore.
L'operatore `def` restituisce il valore *true* se, e solo se, l'operando si valuta in un valore opzionale non vuoto. Se il valore dell'operando non è opzionale, la valutazione solleva un errore.
Per esempio, l'espressione

```
def opt 1 && def opt opt 1 && !def empty opt 1
    && !def empty opt opt 1 && !def empty empty opt 1
```

si valuta in *true*.
L'operatore `get` restituisce il valore contenuto nel valore opzionale ottenuto dalla valutazione del suo operando, se tale valore esiste. La valutazione solleva un errore se l'operando si valuta nel valore opzionale vuoto o in un qualsiasi valore non opzionale.
Per esempio, l'espressione `get opt 1==1 && get opt opt 1==opt 1 && get opt [1]==[1]` si valuta in *true*, mentre la valutazione dell'espressione `get empty opt 1` solleva un errore,

Nota importante: come per le liste, anche per i valori opzionali la semantica degli operatori è funzionale: la valutazione degli operatori **non ha alcun effetto** sugli operandi.
Per esempio, l'esecuzione del seguente programma

```
var l1=[1];
var l2=2::l1;
var o1=opt 1;
var o2=empty o1;
print l1==[1] && o1==opt 1
```

stampa il valore *true*.
La semantica degli statement è quella convenzionale.
`if(`e`){`s_1`}(else{`s_2`})?`: se il valore di e è *true*, allora la sequenza di statement s_1 viene eseguita in un nuovo scope annidato; se il valore di e è *false*, allora la sequenza di statement s_2 viene eseguita in un nuovo scope annidato se la parte `else` è presente, altrimenti l'esecuzione dello statement non ha alcun effetto.
`do{`s`}while(`e`)`: la sequenza di statement s viene eseguita in un nuovo scope annidato almeno una volta; dopo l'esecuzione di s, l'espressione e viene valutata fuori dallo scope annidato; se il suo valore è *true*, allora l'esecuzione dello statement viene ripetuta, se il suo valore è *false*, allora termina.

Nota importante: a ogni iterazione viene creato un nuovo scope annidato per l'esecuzione della sequenza di statement s. Per esempio, l'esecuzione del seguente programma

```
var x=1;
var y=false;
do{x=x+1;var x=true;y=x}while(!(x==5));
print y
```

stampa il valore *true*.

Interfaccia utente

L'interprete deve implementare la seguente interfaccia utente realizzata da linea di comando.

- Il programma da eseguire viene letto dal file di testo *filename* mediante l'opzione `-i` *filename*; il programma viene letto dallo standard input se tale opzione non viene specificata.
- Le stampe del programma in esecuzione vengono salvate sul file di testo *filename* mediante l'opzione `-o` *filename*; le stampe sono visualizzate sullo standard output se tale opzione non viene specificata.

A titolo di esempio, assumendo che `interpreter.Main` sia il nome della classe principale, allora i seguenti casi corrispondono a un corretto utilizzo dell'interfaccia utente:

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

Ogni opzione deve essere necessariamente seguita dal corrispondente nome del file; le opzioni possono essere specificate in qualsiasi ordine e una stessa opzione può essere ripetuta più volte; in tal caso, verrà presa in considerazione solo l'ultima opzione specificata.
L'esecuzione dell'interprete deve aderire al seguente flusso:

- Il programma 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 l'esecuzione passa al punto successivo.
- Il programma viene controllato staticamente; 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 successivo.
- Il programma viene interpretato; 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 comunque 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 l'output prodotto dall'esecuzione del programma interpretato.
Per il formato di stampa dei valori lista e opzionali, fare riferimento al seguente esempio. L'output del programma

```
print [1];
print [1,2];
print opt 1;
print empty opt 1;
print opt opt [1,2];
print empty empty opt [1,2]
```

è

```
[1]
[1, 2]
opt 1
opt empty
opt opt [1, 2]
opt empty
```