Progetto di Linguaggi e Programmazione Orientata agli Oggetti

a.a. 2019/2020

Modificare l'interprete del linguaggio L di riferimento sviluppato negli ultimi tre laboratori (20 aprile, 4 e 11 maggio) 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^{++} permette di gestire anche i literal **`Winter`**, **`Spring`**, **`Summer`** e **`Fall`** di tipo `season`.

Sintassi delle espressioni: in L^{++} è possibile usare l'operatore binario infisso **`<`** di confronto e gli operatori unari infissi **`#`** (numero ordinale associato ai literal di tipo `season`) e **`seasonof`** (literal di tipo `season` associato al corrispondente numero ordinale). Nella grammatica, la definizione del non-terminale `Exp` viene così estesa:

```
Exp ::= ... | Exp<Exp | # Exp | seasonof Exp
```

Nota bene:

- **`seasonof`** è una keyword.
- Le produzioni specificate sopra vanno disambiguate in modo che gli operatori binari associno a sinistra e abbiano meno precedenza degli operatori unari `#` e **`seasonof`**.

  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`**; nella grammatica, la definizione del non-terminale `Stmt` viene così estesa:

```
Stmt ::= ... | for IDENT to Exp {StmtSeq}
```

Nota bene: **`for`** e **`to`** sono keyword.

Complessivamente, le nuove keyword introdotte da L^{++} sono

`Winter`, **`Spring`**, **`Summer`**, **`Fall`**, **`seasonof`**, **`for`**, **`to`**.

Semantica statica

La semantica statica è specificata dal programma OCaml nel file `semantica-statica.ml`.
I literal **`Winter`**, **`Spring`**, **`Summer`** e **`Fall`** hanno tipo primitivo `season`, che è non compatibile con gli altri tipi (`int`, `bool` e tipo prodotto per le coppie).
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 **`#`** è definito solo se l'operando ha tipo `season`; il risultato ha tipo `int`.
L'operatore **`seasonof`** è definito solo se l'operando ha tipo `int`; il risultato ha tipo `season`.
Lo statement **`for`** i **`to`** e{s} è corretto staticamente solo se

- l'identificatore i corrisponde a una variabile di tipo `int` già dichiarata nell'ambiente corrente;
- l'espressione e ha tipo statico `int` nell'ambiente corrente;
- la sequenza di statement s è corretta staticamente nel nuovo ambiente ottenuto da quello corrente aggiungendo un nuovo scope annidato inizialmente vuoto.

Nota importante: le dichiarazioni di variabile nel blocco `{`s`}` sono a un livello di scope più annidato, come accade per lo statement `if-else`. Per esempio, il programma

```
var x=0;
for x to 3{
   print x; // prints 0 1 2 3
   var x=seasonof x;
   print x  // prints Winter Spring Summer Fall
};
print x==4 // prints true
```

è staticamente corretto.

Semantica dinamica

La semantica dinamica è specificata dal programma OCaml contenuto nel file `semantica-dinamica.ml`.
L'operatore di confronto `<` ha una semantica ben definita solo se il secondo operando ha lo stesso tipo del primo. Indipendentemente dal tipo, il confronto v`<`v su uno stesso valore restituisce sempre `false`. La semantica per il tipo `int` è quella convenzionale. Per il tipo `bool`, vale solo la relazione `false<true`, mentre per il tipo `season` valgono le relazioni `Winter<Spring<Summer<Fall`. Una coppia `<<`v_1`,`v_2`>>` è minore di un'altra `<<`v_3`,`v_4`>>` solo se v_1`<`v_3 e v_2`<`v_4.
L'operatore `#` è così definito:
`# Winter` restituisce 0, `# Spring` restituisce 1, `# Summer` restituisce 2 e `# Fall` restituisce 3. In tutti gli altri casi viene sollevata un'eccezione `EvaluatorException` per un errore dinamico di conversione di tipo.
L'operatore `seasonof` è così definito:
`seasonof 0` restituisce `Winter`, `seasonof 1` restituisce `Spring`, `seasonof 2` restituisce `Summer` e `seasonof 3` restituisce `Fall`; in tutti i rimanenti casi in cui l'operando sia un intero, viene sollevata un'eccezione `EvaluatorException` causata dall'eccezione `IndexOutOfBoundsException`. Se l'operando non è di tipo intero, allora viene sollevata un'eccezione `EvaluatorException` per un errore dinamico di conversione di tipo.
La semantica dello statement `for` i `to` e`{`s`}` è così definita:

1. nell'ambiente corrente viene valutato il valore v di i per l'iterazione corrente e il valore M di e corrispondente al valore massimo per i; se v o M non sono di tipo intero, allora viene sollevata un'eccezione `EvaluatorException` per un errore dinamico di conversione di tipo (il tipo di v viene controllato prima del tipo di M);
2. se $v > M$ lo statement `for` termina e viene restituito l'ambiente corrente;
3. se $v <= M$ viene eseguito il prossima ciclo:
   (a) l'ambiente viene aggiornato con un nuovo scope annidato inizialmente vuoto;
   (b) nel nuovo ambiente viene eseguita la sequenza di statement s del blocco;
   (c) l'ambiente viene aggiornato eliminando lo scope più annidato;
   (d) nel nuovo ambiente viene aggiornato il valore della variabile i del for incrementandolo di 1;
   (e) lo statement `for` viene di nuovo eseguito a partire dal punto 1.

Nota importante: all'inizio di ogni singolo ciclo dello statement `for`:

- viene valutata l'espressione e;
- viene creato un nuovo ambiente annidato per eseguire la sequenza di statement s.

Per esempio, il seguente programma:

```
var x=-1;
for x to -2*x{
  var y=x == 0;
  print x;
  print y
};
print x
```

stampa

```
-1
false
0
true
1
```

Per i valori di tipo `season` lo statement **`print`** stampa la corrispondente keyword. Per esempio, il seguente programma

```
var s=0;
for s to 3{
  print seasonof s
}
```

stampa

```
Winter
Spring
Summer
Fall
```

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.