Progetto di Linguaggi e Programmazione Orientata agli Oggetti

a.a. 2016/2017

Implementare in Java un interprete per il linguaggio di programmazione definito più sotto, seguendo le linee guida delle esercitazioni di laboratorio.

Sintassi

La sintassi del linguaggio è definita a partire dal non-terminale `Prog` della seguente grammatica ambigua BNF (dove i simboli terminali sono evidenziati in blu).

```
Prog ::= StmtSeq EOF
StmtSeq ::= Stmt | Stmt;StmtSeq
Stmt ::= ID=Exp | var ID=Exp | print Exp | for ID in Exp {StmtSeq}
       | if (Exp) {StmtSeq} else {StmtSeq} | while (Exp) {StmtSeq}
ExpSeq ::= Exp | Exp,ExpSeq
Exp ::= Exp||Exp | Exp&&Exp | Exp==Exp | Exp<Exp | Exp+Exp | Exp-Exp | Exp*Exp | Exp/Exp
       | !Exp | -Exp | top Exp | pop Exp | push(Exp,Exp) | [ExpSeq] | Exp@Exp | length Exp
       | pair(Exp,Exp) | fst Exp | snd Exp | NUM | BOOL | ID | (Exp)
```

Le categorie lessicali `NUM`, `BOOL` e `ID` sono così specificate:

- i literal interi (`NUM`) includono sia la notazione posizionale in base 10, sia quella in base 8:
  - i literal in base 10 sono costituiti da stringhe non vuote formate da cifre decimali che possono iniziare con la cifra `0` solo se il literal contiene una sola cifra;
  - i literal in base 8 sono costituiti da stringhe di almeno due cifre ottali (ossia da 0 a 7) dove la prima è sempre 0.

  Esempio: `23` è il literal in base 10 che corrisponde al numero ventitré, mentre `023` è il literal in base 8 che corrisponde al numero diciannove.
- i literal booleani (`BOOL`) sono costituiti dalle parole-chiave **`false`** e **`true`**;
- gli identificatori (`IDENT`) sono costituiti da stringhe non vuote dove il primo carattere deve essere una lettera minuscola o maiuscola opzionalmente seguita da una stringa che può contenere solo lettere (minuscole e maiuscole), cifre decimali e il carattere _ (underscore);
- gli elementi lessicali che vengono ignorati sono le sequenze di spazi bianchi (inclusi i terminatori di linea) e i commenti mono-linea che iniziano con la sequenza `//`.

La lista completa delle parole-chiave è la seguente:

`else`, **`false`**, **`for`**, **`fst`**, **`if`**, **`in`**, **`length`**, **`pair`**, **`pop`**, **`print`**, **`push`**, **`snd`**, **`top`**, **`true`**, **`var`**, **`while`**

La seguente tabella specifica l'ordine di precedenza degli operatori binari infissi (in ordine crescente dall'alto al basso, quindi l'operatore `||` ha la precedenza più bassa). Tutti tali operatori associano a sinistra. Ogni operatore unario prefisso ha la precedenza sugli operatori binari infissi.

operatori
`||`
`&&`
`==`
`<`
`@`
`+` `-`
`*` `/`

Semantica statica

Sintassi delle espressioni di tipo:

`Type ::= int | bool | Type list | Type * Type`

I tipi `Type list` e $\texttt{Type}_1$ `*` $\texttt{Type}_2$ rappresentano rispettivamente le liste i cui valori sono di tipo `Type` e le coppie dove il primo valore ha tipo $\texttt{Type}_1$ e il secondo $\texttt{Type}_2$.
Funzioni che definiscono la semantica statica:

$$StaticEnv = \text{ID} \to Type \text{ (funzioni parziali a dominio finito)}$$

$$wf_{Prog} : Prog \to \{ok, error\}$$
$$wf_{Stmt} : Stmt \cup StmtSeq \to StaticEnv \to StaticEnv \cup \{error\}$$
$$wf_{Exp} : Exp \cup ExpSeq \to StaticEnv \to Type \cup \{error\}$$

Definizione delle funzioni di semantica statica
(fare riferimento al materiale relativo al laboratorio 9 per le definizioni delle operazioni sugli ambienti statici)

$wf_{Prog}(stmts) = ok$ se $wf_{Stmt}(stmts)\emptyset \neq error$, $error$ altrimenti
$wf_{Stmt}(stmt; stmts)r = wf_{Stmt}(stmts)r'$ se $wf_{Stmt}(stmt)r = r'$, $error$ altrimenti
$wf_{Stmt}(\textsf{var } id = exp)r = r[ty/id]$ se $wf_{Exp}(exp)r = ty$, $error$ altrimenti
$wf_{Stmt}(id = exp)r = r$ se $r(id) = ty$ e $wf_{Exp}(exp)r = ty$, $error$ altrimenti
$wf_{Stmt}(\textsf{print } exp)r = r$ se $wf_{Exp}(exp)r \neq error$, $error$ altrimenti
$wf_{Stmt}(\textsf{for } id \textsf{ in } exp\ \{stmts\})r = r$ se $wf_{Exp}(exp)r = ty\ list$ e $wf_{Stmt}(stmts)r[ty/id] \neq error$, $error$ altrimenti
$wf_{Stmt}(\textsf{if } (exp)\ \{stmts_1\} \textsf{ else } \{stmts_2\})r = r$ se $wf_{Exp}(exp)r = bool$, $wf_{Stmt}(stmts_1)r \neq error$
e $wf_{Stmt}(stmts_2)r \neq error$, $error$ altrimenti
$wf_{Stmt}(\textsf{while } (exp)\ \{stmts\})r = r$ se $wf_{Exp}(exp)r = bool$, e $wf_{Stmt}(stmts)r \neq error$, $error$ altrimenti
$wf_{Exp}(exp, exps)r = ty$ se $wf_{Exp}(exp)r = ty$ e $wf_{Exp}(exps)r = ty$, $error$ altrimenti
$wf_{Exp}([exps])r = ty\ list$ se $wf_{Exp}(exps)r = ty$, $error$ altrimenti
$wf_{Exp}(\textsf{top } exp)r = ty$ se $wf_{Exp}(exp)r = ty\ list$, $error$ altrimenti
$wf_{Exp}(\textsf{pop } exp)r = ty\ list$ se $wf_{Exp}(exp)r = ty\ list$, $error$ altrimenti
$wf_{Exp}(\textsf{push}(exp_1, exp_2))r = ty\ list$ se $wf_{Exp}(exp_1)r = ty$ e $wf_{Exp}(exp_2)r = ty\ list$, $error$ altrimenti
$wf_{Exp}(exp_1 @ exp_2))r = ty\ list$ se $wf_{Exp}(exp_1)r = wf_{Exp}(exp_2)r = ty\ list$, $error$ altrimenti
$wf_{Exp}(\textsf{length } exp)r = int$ se $wf_{Exp}(exp)r = ty\ list$, $error$ altrimenti
$wf_{Exp}(\textsf{pair}(exp_1, exp_2))r = (ty_1 * ty_2)$ se $wf_{Exp}(exp_1)r = ty_1$ e $wf_{Exp}(exp_2)r = ty_2$, $error$ altrimenti
$wf_{Exp}(\textsf{fst } exp)r = ty_1$ se $wf_{Exp}(exp)r = (ty_1 * ty_2)$, $error$ altrimenti
$wf_{Exp}(\textsf{snd } exp)r = ty_2$ se $wf_{Exp}(exp)r = (ty_1 * ty_2)$, $error$ altrimenti
$wf_{Exp}(exp_1 == exp_2)r = bool$ se $wf_{Exp}(exp_1)r = wf_{Exp}(exp_2)r \neq error$, $error$ altrimenti
$wf_{Exp}(exp_1 < exp_2)r = bool$ se $wf_{Exp}(exp_1)r = wf_{Exp}(exp_2)r = int$, $error$ altrimenti
$wf_{Exp}(exp_1\ op\ exp_2)r = int$ se $wf_{Exp}(exp_1)r = wf_{Exp}(exp_2)r = int$, $op \in \{+, -, *, /\}$, $error$ altrimenti
$wf_{Exp}(exp_1\ op\ exp_2)r = bool$ se $wf_{Exp}(exp_1)r = wf_{Exp}(exp_2)r = bool$, $op \in \{||, \&\&\}$, $error$ altrimenti
$wf_{Exp}(-exp)r = int$ se $wf_{Exp}(exp)r = int$, $error$ altrimenti
$wf_{Exp}(!exp)r = bool$ se $wf_{Exp}(exp)r = bool$, $error$ altrimenti
$wf_{Exp}(id)r = r(id)$ se $id \in dom(r)$, $error$ altrimenti
$wf_{Exp}(num\_lit)r = int$
$wf_{Exp}(bool\_lit)r = bool$

Semantica dinamica

Funzioni che definiscono la semantica dinamica:

$$LocDynEnv = \text{ID} \to Value \text{ (funzioni parziali a dominio finito)}$$

$$Value = \{false, true\} \cup \mathbb{Z} \cup Value^* \cup (Value \times Value)$$

$$DynamicEnv = LocDynEnv^+$$

$$ev_{Prog} : Prog \to \{error\} \cup Value^*$$
$$ev_{Stmt} : Stmt \cup StmtSeq \to DynamicEnv \to (DynamicEnv \times Value^*) \cup \{error\}$$
$$ev_{Exp} : Exp \cup ExpSeq \to DynamicEnv \to Value^* \cup \{error\}$$

Definizione delle funzioni di semantica dinamica
(fare riferimento al materiale relativo al laboratorio 10 per le definizioni delle operazioni sugli ambienti dinamici)

$ev_{Prog}(stmts) = \overline{val}$ se $ev_{Stmt}(stmts)[\emptyset] = (rs, \overline{val})$, $error$ altrimenti
$ev_{Stmt}(stmt; stmts)rs = (rs'', \overline{val} \cdot \overline{val}')$ se $ev_{Stmt}(stmt)rs = (rs', \overline{val})$ e $ev_{Stmt}(stmts)rs' = (rs'', \overline{val}')$, $error$ altrimenti
$ev_{Stmt}(\mathsf{var}\ id = exp)rs = ((new\_fresh\ rs\ val\ id), \epsilon)$ se $ev_{Exp}(exp)rs = val$, $error$ altrimenti
$ev_{Stmt}(id = exp)rs = (rs', \epsilon)$ se $ev_{Exp}(exp)rs = val$ e $rs' = update\ rs\ val\ id$, $error$ altrimenti
$ev_{Stmt}(\mathsf{print}\ exp)rs = (rs, [val])$ se $ev_{Exp}(exp)rs = val$, $error$ altrimenti
$ev_{Stmt}(\mathsf{for}\ id\ \mathsf{in}\ exp\ \{stmts\})rs_0 = (rs_n, \overline{val}_1 \cdot \ldots \cdot \overline{val}_n)$ se $ev_{Exp}(exp)rs_0 = [val_0, \ldots, val_{n-1}]$
$ev_{Stmt}(stmts)\{id \mapsto val_i\}::rs_i = (r_i::rs_{i+1}, \overline{val}_{i+1})$ per ogni $i \in \{0, \ldots, n-1\}$, $error$ altrimenti
$ev_{Stmt}(\mathsf{if}\ (exp)\ \{stmts_1\}\ \mathsf{else}\ \{stmts_2\})rs = (rs', \overline{val})$ se
$ev_{Exp}(exp)rs = true$ e $ev_{Stmt}(stmts_1)\emptyset::rs = (r'::rs', \overline{val})$ oppure
$ev_{Exp}(exp)rs = false$ e $ev_{Stmt}(stmts_2)\emptyset::rs = (r'::rs', \overline{val})$,
$error$ altrimenti
$ev_{Stmt}(\mathsf{while}\ (exp)\ \{stmts\})rs = (rs', \overline{val})$ se
$ev_{Exp}(exp)rs = true, ev_{Stmt}(stmts)\emptyset::rs = (r_1::rs_1, \overline{val}_1)$,
$ev_{Stmt}(\mathsf{while}\ (exp)\ \{stmts\})rs_1 = (r_2::rs_2, \overline{val}_2)$ e $(rs', \overline{val}) = (rs_2, \overline{val}_1 \cdot \overline{val}_2)$ oppure
$ev_{Exp}(exp)rs = false$ e $(rs', \overline{val}) = (rs, \epsilon)$,
$error$ altrimenti
$ev_{Exp}(exp, exps)rs = [val_0, val_1, \ldots, val_n]$ se $ev_{Exp}(exp)rs = val_0$ e $ev_{Exp}(exps)rs = [val_1, \ldots, val_n]$, $error$ altrimenti
$ev_{Exp}([exps])rs = [val_1, \ldots, val_n]$ se $ev_{Exp}(exps)rs = [val_1, \ldots, val_n]$, $error$ altrimenti
$ev_{Exp}(\mathsf{top}\ exp)rs = val_n$ se $ev_{Exp}(exp)rs = [val_1, \ldots, val_n]$ con $n > 0$, $error$ altrimenti
$ev_{Exp}(\mathsf{pop}\ exp)rs = [val_1, \ldots, val_{n-1}]$ se $ev_{Exp}(exp)rs = [val_1, \ldots, val_n]$ con $n > 0$, $error$ altrimenti
$ev_{Exp}(\mathsf{push}(exp_1, exp_2))rs = [val_1, \ldots, val_{n+1}]$ se $ev_{Exp}(exp_1)rs = val_{n+1}$ e $ev_{Exp}(exp_2)rs = [val_1, \ldots, val_n]$,
$error$ altrimenti
$ev_{Exp}(exp_1@exp_2)rs = [val_1, \ldots, val_n, val'_1, \ldots, val'_k]$
se $ev_{Exp}(exp_1)rs = [val_1, \ldots, val_n]$ e $ev_{Exp}(exp_2)rs = [val'_1, \ldots, val'_k]$, $error$ altrimenti
$ev_{Exp}(\mathsf{length}\ exp)rs = n$ se $ev_{Exp}(exp)rs = [val_1, \ldots, val_n]$, $error$ altrimenti
$ev_{Exp}(\mathsf{pair}(exp_1, exp_2))rs = (val_1, val_2)$ se $ev_{Exp}(exp_1)rs = val_1$ e $ev_{Exp}(exp_2)rs = val_2$, $error$ altrimenti
$ev_{Exp}(\mathsf{fst}\ exp)rs = val_1$ se $ev_{Exp}(exp)rs = (val_1, val_2)$, $error$ altrimenti
$ev_{Exp}(\mathsf{snd}\ exp)rs = val_2$ se $ev_{Exp}(exp)rs = (val_1, val_2)$, $error$ altrimenti
$ev_{Exp}(exp_1 == exp_2)rs = (val_1 = val_2)$ se $ev_{Exp}(exp_1)rs = val_1$ e $ev_{Exp}(exp_2)rs = val_2$, $error$ altrimenti
$ev_{Exp}(exp_1 < exp_2)rs = (i_1 < i_2)$ se $ev_{Exp}(exp_1)rs = i_1$ e $ev_{Exp}(exp_2)rs = i_2$, $error$ altrimenti
$ev_{Exp}(exp_1\ op\ exp_2)rs = (i_1\, op\, i_2)$ se $ev_{Exp}(exp_1)rs = i_1$ e $ev_{Exp}(exp_2)rs = i_2$, $op \in \{+, -, *, /\}$, $error$ altrimenti
$ev_{Exp}(exp_1\ op\ exp_2)rs = (b_1\, op\, b_2)$ se $ev_{Exp}(exp_1)rs = b_1$ e $ev_{Exp}(exp_2)rs = b_2$, $op \in \{||, \&\&\}$, $error$ altrimenti
$ev_{Exp}(-exp)rs = -i$ se $ev_{Exp}(exp)rs = i$, $error$ altrimenti
$ev_{Exp}(!exp)rs = \neg b$ se $ev_{Exp}(exp)rs = b$, $error$ altrimenti
$ev_{Exp}(id)rs = val$ se $lookup\ id\ rs = val$, $error$ altrimenti
$ev_{Exp}(num\_lit)rs = i$ se num_lit rappresenta i
$ev_{Exp}(bool\_lit)rs = b$ se $bool\_lit$ rappresenta b

Nota bene, l'uguaglianza tra valori va intesa nel senso più stretto; in particolare due liste sono uguali se hanno lo stesso numero di elementi e gli elementi coincidono per ogni possibile indice valido; due coppie sono uguali se lo sono le corrispondenti componenti.

Interfaccia utente

Implementare una semplice interfaccia utente che permetta di lanciare l'interprete da linea di comando e che aderisca **completamente** alle seguenti specifiche.

- Il programma da interpretare viene caricato da un file di testo se il suo nome viene passato come argomento; altrimenti viene letto dallo standard input.
- Le stampe dell'esecuzione del programma vengono visualizzate su un file di testo, se il suo nome viene passato tramite l'opzione **`-o`**, altrimenti i risultati vengono stampati sullo standard output.

- Il nome dei file di output e/o input vengono passati come argomenti da linea di comando rispettando la sintassi specificata[1] dalla seguente espressione regolare: (**-o** *NomeFile*)? (*NomeFile*)?
  A titolo di esempio, se assumiamo che `interpreter.Main` sia il nome della classe che contiene il metodo `main` dell'applicazione, 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 input.txt
    ```
  - legge il programma dal file `input.txt`, stampa sul file `output.txt`:
    ```
    $ java interpreter.Main -o output.txt input.txt
    ```
- Il flusso di esecuzione dell'interprete è il seguente:
  - Viene effettuata l'analisi sintattica del programma; in caso di errore, sia a livello di tokenizer, sia a livello di parser, 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.
  - Viene effettuato il controllo di tipo sul programma; in caso di errore 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.
  - Viene interpretato il programma; in caso di errore viene stampato sullo standard error il messaggio associato alla corrispondente eccezione sollevata e il programma termina.
  - Qualsiasi tipo di eccezione dovrà essere catturata e gestita stampando su standard error il corrispondente messaggio e terminando l'esecuzione; ogni file aperto dovrà essere chiuso prima che il programma termini, anche in caso di interruzione dovuta a un'eccezione.

L'output deve contenere **esclusivamente** il risultato dell'esecuzione del programma. Per il formato di stampa dei valori fare riferimento ai test disponibili su AulaWeb.

[1] Affinché l'applicazione sia portabile è necessario scaricare la gestione dei file interamente sul package `java.io`; in tale modo, il controllo della sintassi dei nomi dei file verrà automaticamente effettuato dalle opportune classi del package.