# Progetto di Linguaggi e Programmazione Orientata agli Oggetti

a.a. 2018/2019

Modificare l'interprete del linguaggio $L$ di riferimento sviluppato a partire dal laboratorio 9 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 di tipo string e di rappresentare i literal interi senza segno anche in formato esadecimale.

- i literal di tipo string sono rappresentati con la sintassi convenzionale: il carattere `"` delimita una qualsiasi stringa di caratteri di lunghezza arbitraria, con il vincolo che i caratteri `"` e `\` devono essere rappresentati con `\"` e `\\`; per esempio, il literal `"a\\b\""` è sintatticamente corretto e rappresenta la stringa di lunghezza 4, i cui caratteri sono, nell'ordine, `'a'`, `'\'`, `'b'` e `'"'`.
- i literal interi senza segno in formato esadecimale sono rappresentati con la seguente sintassi: iniziano con la cifra `0`, seguita dalla lettera `x` o `X` seguita da una sequenza **non vuota** di cifre esadecimali, ossia, le cifre decimali e le lettere dalla a alla f, minuscole o maiuscole. L'interpretazione della sequenza di cifre esadecimali è quella convenzionale, come implementata nella Java API; per esempio, il literal `0x0aF` rappresenta il numero 175.

**Sintassi delle espressioni:** $L^{++}$ permette di utilizzare gli operatori binari infissi `^` (concatenazione tra stringhe), **`\/`** (unione tra insiemi), **`/\`** (intersezione tra insiemi) e **`in`** (predicato di appartenenza insiemistica) e l'operatore unario prefisso **`#`** (lunghezza di una stringa o cardinalità di un insieme); inoltre è possibile definire literal di tipo insieme; per esempio, l'espressione `{1,2}/\{2,3}` è sintatticamente corretta. La sintassi è definita dalle seguenti produzioni, da aggiungere alla grammatica di $L$ presentata nel testo del laboratorio 9:

```
ExpSeq ::= Exp (, ExpSeq)?
Exp ::= ... | {ExpSeq} | Exp ^ Exp | Exp \/ Exp | Exp /\ Exp | Exp in Exp | # Exp
```

**Nota bene:**

- **`in`** è una keyword.
- Le produzioni specificate sopra vanno disambiguate in modo che gli operatori binari associno a sinistra e abbiano meno precedenza dell'operatore unario `#`.

  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 |
|---|
| && |
| == |
| in |
| \/ |
| /\ |
| ^ |
| + |
| * |

**Sintassi degli statement:** il linguaggio $L^{++}$ include anche lo statement **`while`**, la cui sintassi è definita dalla seguente produzione, da aggiungere alla grammatica di $L$ presentata nel testo del laboratorio 9:

```
Stmt ::= ... | while (Exp) {StmtSeq}
```

**Nota bene:** `while` è una keyword.

### Semantica statica

La semantica statica è specificata formalmente dal programma OCaml nel file `semantica-statica.ml`.
Rispetto al linguaggio sviluppato durante i laboratori, la semantica statica di $L^{++}$ include il tipo `string` per i literal corrispondenti, e il costruttore di tipo unario `set` per i literal di tipo insieme, il cui argomento corrisponde al tipo degli elementi; per esempio, l'espressione `{1,2}` ha tipo `int set`, `{{"one"},{"two"}}` ha tipo `string set set`, mentre `{1,"one"}` non è ben tipata.
L'operatore `^` di concatenazione si aspetta due operandi di tipo `string` e restituisce un risultato di tipo `string`.
Gli operatori `/\` e `\/` si aspettano due operandi di tipo $t$ `set` e restituiscono un risultato di tipo $t$ `set`, per ogni tipo $t$; per esempio, l'espressione `{1,2}/\{2,3}` ha tipo `int set`, mentre `{1,2}/\{"hello"}` non è ben tipata.
L'operatore `in` si aspetta un primo operando di tipo $t$ e un secondo di tipo $t$ `set`, per ogni tipo $t$, e restituisce un risultato di tipo `bool`; per esempio, l'espressione `3 in {1,2}` ha tipo `bool`, mentre `"hello" in {1,2}` non è ben tipata.
L'operatore `#` si aspetta un operando di tipo `string` oppure $t$ `set` per un qualsiasi tipo $t$, e restituisce un risultato di tipo `int`.
Nello statement `while (`$e$`){...}` la guardia $e$ deve avere tipo `bool`.

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

```
let x=0;
while(!(x==9)){
   x=x+1;
   print x;
   let x={x};
   print x
};
print x==9
```

è staticamente corretto.

### Semantica dinamica

La semantica dinamica è specificata formalmente dal programma OCaml contenuto nel file `semantica-dinamica.ml`.
La semantica dell'operatore di concatenazione `^` tra stringhe è quella usuale; inoltre, viene sollevata un'eccezione se uno dei due operandi non si valuta in una stringa.
Un literal di tipo insieme si valuta nell'insieme dei corrispondenti elementi; elementi ripetuti non modificano la semantica, per esempio, `{1,2}` e `{2,1,1,2}` si valutano nello stesso insieme. A livello implementativo è consigliabile usare la classe `java.util.HashSet` per rappresentare i valori di tipo insieme. La semantica degli operatori di unione `\/` e intersezione `/\` è quella convenzionale; inoltre, viene sollevata un'eccezione se uno dei due operandi non si valuta in un insieme.
L'operatore `in` corrisponde all'operatore di appartenenza insiemistica; per esempio `2 in {1,2,3}` e `0 in {1,2,3}` si valutano in `true` e `false`, rispettivamente. Inoltre, viene sollevata un'eccezione se il secondo operando non si valuta in un insieme.
L'operatore `#`, calcola la lunghezza di una stringa o la cardinalità di un insieme, a seconda del tipo dell'argomento. Inoltre, viene sollevata un'eccezione se l'operando non si valuta in una stringa o in un insieme.
L'operatore di uguaglianza `==` si valuta in `false` se i due operandi non sono dello stesso tipo; per esempio, non è possibile che un intero sia uguale a una stringa. Due stringhe sono uguali se rappresentano la stessa successione di caratteri. Due insiemi $s_1$ e $s_2$ sono uguali se tutti i valori in $s_1$ sono contenuti in $s_2$ e viceversa. Per esempio, `{1,2}=={2,1,1,2}` e `{1}=={2,1,1,2}` si valutano in `true` e `false`, rispettivamente.
*Nota importante:* la valutazione di qualsiasi operatore **non ha alcun effetto** sugli operandi. Per esempio, il seguente programma stampa il valore *true*.

```
let s1={1}; let s2={1}; let t1={2}; let t2={2};
let s=s1\/t1;
print s1==s2 && t1==t2
```

La semantica dello statement `while` è quella convenzionale.
*Nota importante:* all'inizio di ogni iterazione si entra in un nuovo scope annidato per l'esecuzione del corpo dello statement `while` dal quale si esce alla fine dell'esecuzione del blocco stesso. Per esempio, il seguente programma stampa il valore *true*.

```
let x=1; let y=0; let s={0}/\{1};
while(!(x==5)){
    x=x+1;
    let y=y+1;
    s=s\/{y}
};
print s=={1}
```

Lo statement `print` stampa un valore sullo standard output seguito dal carattere new-line.
I valori interi vengono sempre stampati nell'usuale formato in base 10, anche quando rappresentati da literal esadecimali. Per i valori di tipo stringa, viene stampata la stringa corrispondente, senza delimitatori e caratteri di escape. I valori di tipo insieme vengono stampati usando le usuali parentesi graffe come delimitatori e separando gli elementi con la stringa `", "`; nessun spazio bianco viene inserito durante la stampa di un valore di tipo insieme, se non immediatamente dopo la virgola. Gli elementi dell'insieme vengono stampati sulla stessa linea, senza un ordine predefinito e senza ripetizioni.
Per esempio, il seguente programma

```
print 0xff;
print "a\"a\\a";
print { 1 }/\{ 2 };
print { 1 };
print { 1 , 2 , 2 , 1 }
```

stampa

```
255
a"a\a
{}
{1}
{1, 2}
```

### Interfaccia utente

L'implementazione dell'interprete deve essere conforme alla seguente interfaccia utente realizzata tramite linea di comando.

- Il programma da eseguire viene letto dal file di testo *filename* con l'opzione `-i` *filename*; il programma viene letto dallo standard input se tale opzione non viene specificata.
- L'output del programma in esecuzione è il file di testo *filename* se è stata specificata l'opzione `-o` *filename*; altrimenti è lo standard output.
- Il type-checking non viene eseguito se viene specificata l'opzione `-ntc` (abbreviazione di no-type-checking).

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

- legge il programma dallo standard input, stampa sullo standard output:
  `$ java interpreter.Main`
- legge il programma dallo standard input, stampa sullo standard output, non esegue il type-checking:
  `$ java interpreter.Main -ntc`
- 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`
- legge il programma dal file `input.txt`, stampa sul file `output.txt`, non esegue 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 tal caso, verrà presa in considerazione solo l'ultima opzione specificata. Ogni opzione `-i` o `-o` deve essere necessariamente seguita dal corrispondente nome del file.
L'interprete deve essere conforme al seguente flusso di esecuzione:

1. 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, 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 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 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.