Progetto del Corso di
Laboratorio di Linguaggi
Anno Accademico 2021/2022

Linguaggio HaveFun

Assegnamento
L'obiettivo del progetto è l'implementazione del nuovo linguaggio HaveFun. Per fare questo, si estenda l'interprete Imp presentato a lezione e scaricabile dall'elearning seguendo il percorso /Project/Base /imp.zip con le seguenti, nuove definizioni:

■ **Struttura dei Programmi.** Un programma HaveFun avrà la seguente struttura:

$$fun_1 \ldots fun_n\ com$$

ovvero,

1. una sequenza arbitrariamente lunga e potenzialmente vuota di definizioni di funzione (descritte al punto successivo); seguite da
2. un singolo comando (non opzionale).

Fondamentalmente, l'unica differenza strutturale tra i programmi scritti in Imp e HaveFun è la possibilità del secondo di definire una serie di funzioni all'inizio dei programmi.

■ **Definizione di Funzione.** Una definizione di funzione ha la seguente sintassi:

$$\text{fun } id_0\ (id_1, \ldots, id_n)\ \{\ com; \text{ return } exp\ \}$$

Si presti attenzione ai seguenti punti:

- sono ammesse funzioni con **zero** parametri, ad esempio

  ```
  fun f() { return 1 }
  ```

- il corpo (ovvero, *com*) della funzione è **opzionale** (vedi esempio precedente);

- se il corpo della funzione è presente, allora dovrà essere obbligatoriamente seguito da un **punto e virgola**;

- il return deve comparire **obbligatoriamente e solo** come ultimo statement durante la definizione di una funzione; e

- non sono ammesse funzioni che non ritornano nessun valore, ad esempio

  ```
  fun f() { x = 0; y = 1; return }
  ```

  non sarà un programma valido. In altre parole, *exp* non è opzionale.

L'interpretazione di una dichiarazione di funzione ha l'obiettivo di creare un bind tra il nome logico della funzione id_0 e la sua implementazione, in modo tale che sia poi richiamabile dal codice (vedi il punto successivo). Per fare questo, dovrete modificare/estendere in modo opportuno la configurazione dell'interprete in modo che tenga traccia di tutte queste definizioni.

> **Hint:** Implementare una classe FunValue estendente Value che memorizza i seguenti aspetti: nome della funzione, lista dei nomi dei parametri, corpo della funzione (ovvero, il ComContext), ed espressione di ritorno (ExpContext).

Inoltre, la visita di una definizione di funzione dovrà assicurarsi che

- la funzione id_0 **non sia già definita**. In questo caso, stampate a video un errore ed interrompete l'esecuzione. Ad esempio,

  ```
  fun f(x, y, z) {
     return x
  }

  fun f(x, y, z) {
     return y
  }
  ```

  stamperà a video

  ```
  Fun f already defined
  @5:0
  ```

- i parametri formali della stessa funzione (ovvero, $id1, \ldots, idn$) devono essere tutti **diversi** tra loro. Ad esempio,

  ```
  fun f(x, y, x) {
     return x
  }
  ```

  stamperà a video

  ```
  Parameter name x clashes with previous parameters
  @1:0
  ```

■ **Chiamata di Funzione.** La chiamata di una funzione sarà un'**espressione** con la seguente sintassi:

$$id\ (exp_1, \ldots, exp_n)$$

In fase di visiting, dovrete prestare attenzione ai seguenti aspetti:

- assicurarsi che la funzione *id* **sia stata definita**, in caso contrario ritornare un errore. Ad esempio,

```
fun f(x, y, z) {
   return x
}

out(g(4))
```

stamperà

```
Function g used but never declared
@5:4
```

- assicurarsi che il numero di argomenti con cui la funzione viene chiamata **corrisponda** al numero di parametri formali con cui è stata definita. Ad esempio,

```
fun f(x, y, z) {
   return x
}

out(f(4))
```

stamperà

```
Function f called with the wrong number of arguments
@5:4
```

Un aspetto molto importante riguarda le **regole di scoping**. Ogni funzione chiamata vede nel suo scope i suoi parametri formali ed eventualmente le **variabili globali** (vedere la sezione successiva). Le funzioni non hanno modo nè di accedere nè di modificare sia le variabili non globali esterne alle funzioni che le variabili dichiarate in altre funzioni. Ovviamente il corpo della funzione chiamata può definire nuove variabili utilizzabili all'interno della funzione stessa.
Esempi:

- Programma:

```
fun f(x) {
   return y
}

y = 5;
out(f(y))
```

Output:

```
Variable y used but never instantiated
@2:9
```

- Programma:

```
fun f(x) {
   return y.g
}
```

```
global y = 5;
y = 3;
out(f(y.g))
```

Output:

```
5
```

- Programma:

```
fun f(x) {
   y = 0;
   out(y);
   return y
}

y = 5;
out(f(y));
out(y)
```

Output:

```
0
0
5
```

- Programma:

```
fun f(x) {
   y = 0;
   out(y);
   out(y.g);
   return y
}

y = 5;
global y = 3;
out(f(y));
out(y)
```

Output:

```
0
3
0
5
```

- Programma:

```
fun f(x) {
   y = 0;
   r = g(y);
```

```
    out(y);
    out(r);
    return y
}

fun g(x) {
    y = 10;
    out(y);
    return y
}

y = 5;
out(f(y));
out(y)
```

Output:
10
0
10
0
5

■ Variabili globali.

Si differenziano per tre peculiarità:

1. Necessitano della keyword "*global*" nella dichiarazione, esempio:

$$\ldots \textit{global } x = 50; \ldots$$

2. Le funzioni hanno modo di accedere ad esse e di modificarne il valore.

3. In caso di utilizzo di una variabile globale è necessario aggiungere " *.g* " all'identificatore della variabile, esempio:

$$\ldots x.g + 10; \ldots$$

NB: possono esistere variabili locali aventi lo stesso identificatore di una qualunque variabile esterna alla funzione considerata. Inoltre, le variabili che non possono essere modificate all'interno delle funzioni possono avere lo stesso identificatore di quelle globali.

■ Non determinismo.

Aggiungere un comando " nd " che permetta la **scelta non deterministica** di due comandi, con la seguente sintassi:

$$\{\ com_1\ \}\ \text{nd}\ \{\ com_2\ \}$$

Esempi:

- Programma:

  ```
  fun f(x) {
     y = x + 10;
     return y
  }

  y = 5;

  { out(f(y)) } nd { out(y) }
  ```

  Possibili Output:

  15

  Oppure

  5

- Programma:

  ```
  fun f(x) {
     y = x + 10;
     return y
  }

  y = 5;
  global y = 5;

  { out(f(y)) } nd { { out(y) } nd { out(y.g + 10) } }
  ```

  Possibili Output:

  15

  Oppure

  5

Linee Guida

Creare un nuovo progetto IntelliJ di nome havefun a partire dal codice (dopo averne fatto il refactoring) contenuto in imp.zip. Lo svolgimento dell'elaborato può essere suddiviso in due fasi:

Estensione della grammatica: per ognuno dei costrutti da implementare, si estenda la grammatica del linguaggio in modo consistente con le regole sintattiche mostrate nella sezione precedente.

Implementazione della semantica dinamica: si esegua l'override dei nuovi metodi generati da ANTLR in seguito alla modifica della grammatica HaveFun.g4 nella classe IntHaveFun al fine di implementarne la semantica dinamica.

È altamente consigliato utilizzare jdk18, nel caso ci siano problemi, scrivete un'e-mail cosi cerchiamo di trovare una soluzione.

I progetti verranno corretti in ambiente "tux" (linux), perciò se nel progetto avete inserito dei path controllate che utilizzino "/" e non "\".

Consegna del progetto:

1. Creare una cartella condivisa su OneDrive,
2. Caricare un archivio (nome cognome.tar) contenente tutti i file del progetto (compresa la libreria di ANTLR e le due cartelle contenenti i test),
3. Inviare un'e-mail contenente: nome, cognome, matricola, link alla cartella, os utilizzato, jdk utilizzato, e versione plugin ANTLR utilizzata.

Testing

Una volta terminata l'implementazione, si eseguano dei test sia rispetto a programmi corretti che non per verificarne la correttezza dell'implementazione stessa. In particolare, una serie di programmi sarà caricata tra qualche giorno su Moodle al percorso /Project/Test. I programmi all'interno della sottocartella Well sono programmi corretti che devono essere eseguiti dall'interprete senza nessun errore. Invece, i programmi all'interno della sottocartella Bad sono programmi non corretti che devono stampare a video un errore durante la loro esecuzione.

NB IL PROGETTO CONSEGNATO DEVE PASSARE TUTTI I TEST.

In caso di dubbi, perplessità, e per la consegna del progetto, l'indirizzo e-mail di riferimento è il seguente: marco.vedovato@studenti.univr.it