

Progetto del Corso di Laboratorio di Linguaggi

Anno Accademico 2020/2021

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 le seguente struttura:

$$fun_1 \dots fun_n \text{ com}$$

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:

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

Si presti attenzione ai seguenti punti:

- sono ammesse funzioni con **zero** parametri, ad esempio
- 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 estendendo 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, id_1, \dots, id_n) 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:

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

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 **solamente** i suoi parametri formali. Le funzioni non hanno modo nè di accedere nè di modificare sia le variabili globali (esterne alle funzioni) sia 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) {
  y = 0;
  out(y);
  return y
}

```

```

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

```

Output:

```

0
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

```

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.

Testing

Una volta terminata l'implementazione, si eseguano dei test sia rispetto a programmi ben tipati che non per verificarne la correttezza. In particolare, una serie di programmi sono già stati caricati sull'elearning al percorso `/Project/Test`. I programmi all'interno della sottocartella `Well-Typed` sono programmi ben tipati che devono essere eseguiti dall'interprete senza nessun errore. Invece, i programmi all'interno della sottocartella `Bad-Typed` sono programmi mal tipati che devono stampare a video un errore durante la loro esecuzione.