Corso di Linguaggi e Traduttori
Anno Accademico 2016–2017
Prof. Giovanni Pighizzini

Progetto d'esame

Scopo del progetto è la costruzione di un *interprete* o di un *compilatore* per la valutazione di programmi sorgenti contenenti espressioni aritmetiche su interi e con identificatori, come specificato qui di seguito.

Le espressioni sono basate sulle quattro operazioni aritmetiche, più l'operazione di resto della divisione. Possono inoltre contenere gli operatori di incremento e decremento, prefissi o postfissi, e l'operatore di assegnamento che, come nel linguaggio Java, producono effetti collaterali, oltre al risultato. Tutte le operazioni si svolgono tra numeri interi e *producono sempre un risultato intero*.

Un *programma sorgente* è costituito da una *sequenza non vuota di espressioni* e da una *specifica di ripetizione*, opzionale, che permette di ripetere il calcolo delle espressioni. Alcuni esempi di programmi sono riportati più avanti, insieme ad alcuni esempi di esecuzione.

Illustriamo prima di tutto il caso in cui non sia presente la specifica di ripetizione. Durante l'esecuzione, il programma deve prima di tutto richiedere all'utente il valore di ciascuna variabile, visualizzandone il nome seguito da un punto interrogativo. Il programma deve poi visualizzare, uno per riga, i risultati delle espressioni. Nel caso si tenti di effettuare una divisione per zero, al posto del risultato dell'espressione si dovrà visualizzare `Err` (eventuali effetti collaterali dovuti a sottoespressioni già calcolate prima della divisione potrebbero comunque avere modificato il valore di alcune variabili).

La *specifica di ripetizione* permette di indicare la ripetizione del calcolo delle espressioni ed è costituita da due parti:

- La parola riservata `while`, seguita da un'espressione.
- Una parte opzionale introdotta dalla parola riservata `keeping`, seguita da un elenco di variabili.

Se il risultato dell'espressione è zero, il programma termina, altrimenti dopo avere richiesto all'utente i valori per le variabili (eccetto quelle indicate nell'elenco dopo `keeping`, se presente), l'esecuzione prosegue valutando nuovamente le espressioni.

La grammatica del linguaggio è riportata qui di seguito. Per le precedenze ed associatività degli operatori si faccia riferimento a quelle del linguaggio Java (si ricordi che gli operatori binari sono associativi a sinistra, con l'eccezione dell'operatore di assegnamento che è associativo a destra).

Modificato il 13 giugno 2017

La grammatica

programma → *seqEspr ripetizione*

seqEspr → *espressione* `;`
| *seqEspr espressione* `;`

espressione → *numero*
| *identificatore*
| *espressione* `+` *espressione*
| *espressione* `-` *espressione*
| *espressione* `*` *espressione*
| *espressione* `/` *espressione*
| *espressione* `%` *espressione*
| `-` *espressione*
| `+` *espressione*
| `(` *espressione* `)`
| `++` *identificatore*
| `--` *identificatore*
| *identificatore* `++`
| *identificatore* `--`
| *identificatore* `=` *espressione*

ripetizione → *direttivaWhile direttivaKeep*
| ε

direttivaKeep → `keeping` *seqIdentificatori*
| ε

seqIdentificatori → *identificatore*
| *seqIdentificatori* `,` *identificatore*

direttivaWhile → `while` *espressione*

identificatore → sequenza di lettere e cifre che inizia con una lettera

numero → sequenza non vuota di cifre

Come in Java o C i ritorni a capo sono caratteri si spaziatura. Una espressione potrebbe essere scritta su più righe. Più espressioni potrebbero essere scritte su una sola riga. L'intero programma potrebbe essere scritto su un'unica riga.

Alcune osservazioni

- Attenzione a non rivalutare più volte una stessa espressione a fronte di una sola occorrenza nel sorgente. Ad esempio, nel calcolo di `a % b` le espressioni `a` e `b` devono essere valutate una volta sola.
- Per gli operatori di incremento e decremento, si presti attenzione alla differenza tra la forma prefissa e la forma postifissa.

Alcuni esempi

Testo sorgente

```
x * (2 + x);
x + y;
y;
```

Esempi di esecuzione

```
x? 2
y? 1
8
3
1
```

```
x + 2 * y - z;
3 * x - 1;
z + w--;
x + (x = w);
x;
```

```
x? 2
y? 1
z? 4
w? 2
0
5
6
3
1
```

```
y++ / (x - 2);
x + y;
```

```
x? 1
y? 2
-2
4
```

```
x? 2
y? 2
Err
5
```

```
x * y;
y--;
while y keeping x, y
```

```
x? 3
y? 2
6
2
3
1
```

```
x * y;
y--;
while y keeping y
```

```
x? 3
y? 2
6
2
x? 8
8
1
```

```
x * y;
while y-- keeping y
```

```
x? 3
y? 2
6
x? 8
8
x? 100
0
```

```
dividendo; divisore;
while (dividendo = dividendo % divisore) +
      (divisore  = divisore + dividendo) -
      (dividendo = divisore - dividendo) -
      (divisore  = divisore - dividendo)
      keeping dividendo, divisore
```

```
dividendo? 24
divisore? 9
24
9
9
6
6
3
```

```
dividendo? 945
divisore? 75
945
75
75
45
45
30
30
15
```

Cosa è richiesto

Realizzare *a propria scelta* un interprete o un compilatore. Si suggerisce di ispirarsi agli esempi presentati a lezione.

- Scrivete un analizzatore lessicale e un analizzatore sintattico per il linguaggio, servendovi degli strumenti presentati a lezione o di strumenti differenti *purché adeguatamente documentati e concordati preventivamente.*
- Scrivete una classe di prova per l'analizzatore lessicale che elenchi i token man mano inseriti.
- Nel caso si scriva un compilatore, questo dovrà generare codice per la macchina a stack presentata a lezione. Per generare il codice e per eseguirlo si utilizzino le classi `Codice.java` e `Macchina.java` che NON DEVONO essere modificate. (Sono possibili altre soluzioni, purché concordate preventivamente con il docente.)
- In caso di errore in compilazione l'applicazione può terminare l'esecuzione, fornendo un breve messaggio relativo al problema riscontrato. Un modo rudimentale per ottenere queste informazioni relative agli errori sintattici, consiste nell'inserire nel file di specifica sintattica di CUP il seguente codice:

```
parser code{:
  /* Ridefinizione del metodo che visualizza i messaggi di errore */
  public void unrecovered_syntax_error(Symbol cur_token)
                                       throws java.lang.Exception {
    Scanner sc = (Scanner) getScanner();  //riferimento all'analizzatore
                                          //lessicale in uso
    report_fatal_error("Errore di sintassi alla riga "  +
           sc.currentLineNumber() + " leggendo " + sc.yytext(), null);
    //numero della riga in esame e testo corrispondente al token corrente
  }
:}
```

e nel file di specifica lessicale di JFlex (parte opzioni e dichiarazioni) il seguente:

```
%{
  public int currentLineNumber() {
    return yyline + 1;
   }
%}
%line
```

Si deve consegnare:

1. una breve descrizione delle classi utilizzate e dell'organizzazione della symbol table;
2. tutti file sorgenti scritti per la risoluzione del problema (file di specifica lessicale, file di specifica sintattica, classi o altro codice scritto), in forma elettronica, con un indice degli stessi;
3. alcuni esempi di compilazione *significativi* (sorgente e codice generato, *in forma leggibile*).

È possibile svolgere il progetto in gruppi di due persone.

Il progetto è valido per l'anno accademico 2016/2017 e deve essere consegnato almeno dieci giorni prima della data concordata per la prova orale. (È necessario iscriversi tramite SIFA all'appello del mese in cui si presenta il progetto).