Corso di Linguaggi e Traduttori
Anno Accademico 2017–2018
Prof. Giovanni Pighizzini

Progetto d'esame

Scopo del progetto è la costruzione di un *compilatore* per il linguaggio descritto qui di seguito.

Variabili

Il linguaggio prevede variabili di tipo intero, dichiarate implicitamente al primo utilizzo. Un identificatore di variabile è una sequenza di lettere e cifre che inizia con una lettera e che non sia una delle parole riservate del linguaggio (indicate man mano nel seguito).

Stabilite una strategia per le variabili non inizializzate (esempi: inizializzazione automatica a un valore di default o a un valore random, oppure, come in Java, segnalazione di errore in compilazione quando vi sia un'istruzione che tenti di utilizzare il valore di una variabile non inizializzata da un'istruzione precedente).

Letterali interi

Sono sequenze non vuote di cifre decimali che denotano numeri interi in notazione decimale e sequenze non vuote di cifre esadecimali, precedute da `0x`, che denotano numeri interi in notazione esadecimale.

Espressioni

Sono costruite utilizzando gli operatori sottoindicati e le parentesi tonde, a partire da variabili e letterali interi, seguendo la semantica e le regole di precedenza e associatività del linguaggio Java. Tutte le operazioni avvengono secondo l'aritmetica intera e producono risultati interi.

- Operatori binari `+`, `-`,`*`, `/`, `%`, `=` e operatori unari `+` e `-`.
- Operatore condizionale (ternario) `?:`.
- Operatore di lettura `input` (parola riservata). Questo operatore fornisce come risultato un valore intero letto da input, che potrà essere assegnato a una variabile o utilizzato nel calcolo di un'espressione. L'operatore può essere seguito da una stringa che verrà visualizzata sul monitor prima della lettura del dato. L'utente fornisce l'input esclusivamente in notazione decimale.

Quando un'espressione è utilizzata come *condizione* (ad esempio come primo operando nell'operatore condizionale o nell'istruzione di ripetizione, descritta più avanti), un valore diverso da 0 indica *vero*, mentre 0 indica *falso*, come nel linguaggio C. Si noti che per calcolare il risultato di un'operazione, devono essere calcolati prima i valori degli operandi, *salvo nel caso dell'operatore condizionale*, dove si calcola il valore del primo operando e, in base ad esso, uno solo tra il secondo e il terzo (questa modalità di calcolo produce un effetto differente, rispetto a calcolare tutti gli operandi, nel caso il secondo o il terzo operando producano effetti collaterali, dovuti a operatori di assegnamento o lettura).

Pubblicato il 30 maggio 2018 – Modificato il 12 giugno 2018

Istruzione di Assegnamento

L'*istruzioni di assegnamento* ha la seguenti forma

identificatore `=` *espressione*

Si osservi che, come in Java e in C, l'operatore di assegnamento può fare parte di espressioni. Pertanto è possibile scrivere un'istruzione come

```
a = a + (a = a * a)
```

che, salvo casi particolari, produce un effetto diverso rispetto a

```
a = (a = a * a) + a
```

Istruzioni di Scrittura

L'*istruzione di scrittura* ha le forme

`output` *stringa espressione*
`output` *stringa*
`output` *espressione*

dove `output` è una parola riservata. Nella prima forma viene visualizzata sul monitor la stringa indicata seguita dal risultato dell'espressione, in notazione decimale. Nelle altre due forme viene visualizzata solo la stringa o solo il risultato dell'espressione. *In ogni caso* il cursore non viene riportato a capo. Pertanto successive istruzioni di scrittura proseguiranno a scrivere sulla medesima riga.

Per spostare il cursore all'inizio della riga successiva si utilizza l'istruzione

```
newLine
```

Ripetizione

La seguente istruzione permette di ripetere una sequenza di istruzioni sulla base del valore di un'espressione:

`loop` *espressione*
  seqIstruzioni
`endLoop`

dove `loop` ed `endloop` sono parole riservate, *seqIstruzioni* è una sequenza di istruzioni qualunque (incluse istruzioni `loop`). La semantica è analoga a quella del ciclo `while` in Java o C. In particolare se alla prima valutazione l'espressione vale 0, la sequenza di istruzioni non viene eseguita.

Si noti che, come nel linguaggio Java, le dichiarazioni seguono la struttura statica del programma e non il flusso di esecuzione, come nel seguente esempio:[1]

```
x = input "Primo numero? "
y = input "Secondo numero? "
loop y
  resto = x % y
  x = y
  y = resto
endloop
output "mcd = " x
newLine
```

Se il corpo del ciclo non viene eseguito, la variabile `resto` alla fine dell'esecuzione non risulta inizializzata. In questo caso ciò non è un problema, in quanto non si tenta di utilizzarne il valore.

[1] Dunque una variabile che appare per la prima volta all'interno di un ciclo è dichiarata (implicitamente) un'unica volta, indipendentemente da quante volte verrà eseguito il corpo del ciclo.

Programma

Un *programma sorgente* è costituito da una *sequenza non vuota di istruzioni*. Vi sono alcuni vincoli sul formato delle istruzioni (che *a differenza* di Java e C *non* sono delimitate dal punto e virgola).

- Ogni istruzione semplice (cioè di assegnamento o di scrittura) viene scritta su un'unica riga. Per indicare la continuazione di un'istruzione sulla riga successiva si deve utilizzare il carattere `&`. Ad esempio, al posto di

  ```
  int resto = x % y
  ```

  si può scrivere

  ```
  int resto = &
  x &
  % y
  ```

- Non si possono avere più istruzioni sulla stessa riga.
- Si possono lasciare righe vuote o contenenti solo commenti.
- L'apertura e la chiusura di una ripetizione, cioè `loop` *espressione* ed `endloop`, devono trovarsi ciascuna su una sola riga, senza altre istruzioni, come nell'esempio precedente. Anche in questo caso è possibile proseguire sulla riga successiva con il carattere `&` (ciò può risultare utile, ad esempio, se l'espressione è molto lunga).

Commenti

È possibile inserire commenti, che iniziano con la coppia di caratteri `//` e si estendono sino a fine riga. È possible che un commento occupi un'intera riga o che si trovi dopo il carattere di prosecuzione `&`, come nei seguenti esempi:

```
//calcolo del resto
resto = &  //segue divisione intera
x % y
```

oppure

```
//calcolo del resto
resto = &
& //segue resto di divisione intera
x % y
```

Il seguente esempio non è corretto

```
//calcolo del resto
resto = &
//segue resto di divisione intera
x % y
```

in quanto sulla terza riga manca la prosecuzione dell'istruzione `resto =`. Anche

```
//calcolo del resto
resto = &
//segue resto di divisione intera &
x % y
```

non è corretto (il carattere `&` sulla seconda riga non ha il significato di prosecuzione, poiché fa parte del commento).

La grammatica

programma → *seqIstruzioni*

seqIstruzioni → *istruzione cr*
| *seqIstruzioni istruzione cr*

istruzione → *assegnamento*
| *scrittura*
| *ripetizione*

assegnamento → *identificatore* `=` *espressione*

scrittura → `output` *stringa espressione*
| `output` *stringa*
| `output` *espressione*
| `newLine`

ripetizione → `loop` *espressione cr seqIstruzioni* `endLoop`

espressione → *numero*
| *identificatore*
| *espressione* `+` *espressione*
| *espressione* `-` *espressione*
| *espressione* `*` *espressione*
| *espressione* `/` *espressione*
| *espressione* `%` *espressione*
| `-` *espressione*
| `+` *espressione*
| `(` *espressione* `)`
| *identificatore* `=` *espressione*
| *espressione* `?` *espressione* `:` *espressione*
| `input`
| `input` *stringa*

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

numero → sequenza non vuota di cifre decimali
| `0x` seguita da una sequenza non vuota di cifre esadecimali

stringa → sequenza non vuota di caratteri delimitata da virgolette

cr → fine riga

Alcune osservazioni

- Nella precedente grammatica non sono indicati né i commenti, né il carattere di prosecuzione sulla riga successiva `&`. *È opportuno gestire questi aspetti a livello di analisi lessicale.* Anche la gestione dei letterali in notazione esadecimale può essere effettuata durante l'analisi lessicale.
- 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 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).

Alcuni esempi di sorgenti

```
//massimo comun divisore
x = input "Primo numero? "
y = input "Secondo numero? "
loop y
   resto = x % y
   x = y
   y = resto
endloop
output "mcd = " x
newLine
```

```
//moltiplicazioni
n = input "inserisci il massimo numero da considerare "
i = 1
loop n - i + 1
   j = 1
   loop n - j + 1
      output i
      output " * " j
      output " = " i * j
      newLine
      j = j + 1
   endLoop
   i = i + 1
endLoop
```

```
//pari o dispari
r1 = r2 = input "numero? " % 2
loop r1
   output "dispari"
   r1 = 0
end
loop 1 - r2
   output "pari"
   r2 = 1
endLoop
newLine
```

```
//minimo (tra numeri non negativi)
x = input "primo numero? "
y = input "secondo numero? "
cont = 0
loop x * y
   x = x - 1
   y = y - 1
   cont = cont + 1
end
output cont
newLine
```

Cosa è richiesto

Realizzate un compilatore per il linguaggio descritto precedentemente. Si suggerisce di ispirarsi agli esempi presentati a lezione.

- Scrivete un analizzatore lessicale e un analizzatore sintattico, 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.
- Per generare ed eseguire il codice per la macchina a stack presentata a lezione 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
```

Cosa 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 2017/2018 e deve essere consegnato almeno dieci giorni prima della data concordata per la prova orale.
Per sostenere l'esame è necessario iscriversi preventivamente tramite SIFA.