Linguaggi e Traduttori II

Progetto d'esame a.a. 2008/09

Scopo del progetto è la costruzione di un compilatore per il linguaggio descritto di seguito (la grammatica del linguaggio è riportata alla fine del documemto).

Variabili e tipi

- Il linguaggio prevede variabili di tipo intero e di tipo array di interi.
- Un identificatore di variabile è una sequenza di lettere e cifre che inizia con una lettera.
- Le variabili devono essere *dichiarate esplicitamente* prima di essere utilizzate. Inoltre, nel caso degli array, deve essere indicato, mediante un letterale intero, il numero di elementi. Una stessa variabile non può essere dichiarata più di una volta.

  Per convenzione l'indice zero indica il primo elemento contenuto in un array. La parola riservata `var` introduce la dichiarazione di una o più variabili di tipo intero, mentre la parola riservata `array` introduce la dichiarazione di uno o più array. Le dichiarazioni si chiudono con un punto e virgola. Si utilizza la virgola per separare più variabili nella stessa dichiarazione. Ad esempio, si considerino le seguenti righe di codice:

  ```
  var x;
  array a[2], z[4];
  var w, k, y;
  ```

  Nella prima e nella terza riga vengono dichiarate le variabili `x`, `w`, `k` e `y` di tipo intero, nella seconda gli array `a` e `z` contenenti interi. L'array `a` è costituito da due elementi (indici `0` e `1`), l'array `z` da `4` (indici da `0` a `3`).[1]

Espressioni e condizioni

Il linguaggio prevede le usuali espressioni aritmetiche su interi, costruite a partire da variabili intere, elementi di array, costanti intere, con le operazioni di somma, sottrazione, moltiplicazione, divisione intera e resto della divisione (indicate rispettivamente con i simboli `+, -, *, /, %`), e le parentesi tonde. Un esempio di espressione corretta è `x+z[z[x-1]*2] /(pippo+y)`.

Le condizioni possono essere ottenute mediante gli operatori di confronto `<`, `<=`, `>`, `>=`, `==`, `!=`, applicati a espressioni intere, e combinando condizioni mediante gli operatori `&&`, `||` e `!`.

Per il significato degli operatori e le loro precedenze e associatività si faccia riferimento alle regole del linguaggio Java. In particolare, si richiede che la valutazione degli operatori `&&` e `||` sia lazy.

Istruzioni

- Assegnamento:
  È possibile assegnare il risultato di un'espressione intera a una variabile intera o a un elemento di un array.
  È inoltre possibile *assegnare un array a un altro array*: in questo caso gli elementi vengono copiati nelle posizioni corrispondenti, terminando quando si raggiunge la capacità minima dei due array coinvolti. Ad esempio, con riferimento alle dichiarazioni precedenti, l'assegnamento `a = z;` equivale alla sequenza di istruzioni: `a[0] = z[0]; a[1] = z[1];` mentre l'assegnamento `z = a;` equivale a `z[0] = a[0]; z[1] = a[1];`

[1] Nulla vieta di avere array di zero o un elemento. Questi ultimi non devono essere confusi con le variabili intere.

- Selezione:
  Istruzione `if` con parte `else` opzionale (come in Java)

- Iterazione:
  Istruzione `while` (come in Java) e istruzione `for`. L'istruzione `for` prevede l'indicazione di una variabile e di due espressioni:

  ```
  for (x, espressione1, espressione2)
    ...
  ```

  L'istruzione interna dovrà essere eseguita con `x` che vale inizialmente quanto `espressione1`, incrementando ad ogni iterazione il valore di `x`, terminando quando il valore di `x` supera il valore iniziale di `espressione2`.
  Note:

  - La variabile `x`, che deve essere stata dichiarata in precedenza, può essere una variabile intera o un elemento di un array, come `a[i]`. Si osservi che la variabile *deve rimanere la stessa* durante l'esecuzione del ciclo. Pertanto, se la variabile indicata è `a[i]` e `i` contiene `2` nel momento in cui inizia l'esecuzione del ciclo, la variabile del ciclo sarà `a[2]` durante tutta la sua esecuzione, anche se il valore di `i` dovesse essere modificato.
  - Se inizialmente il valore di `espressione1` supera quello di `espressione2` il ciclo termina immediatamente, senza che sia eseguita l'istruzione interna.
  - Per questioni di efficienza `espressione2` deve essere valutata esclusivamente all'inizio dell'esecuzione del ciclo e *non a ogni iterazione*. Pertanto se i valori coinvolti nell'espressione cambiano durante l'esecuzione del ciclo, il valore di `espressione2` considerato ai fini della terminazione dovrà essere quello valutato inizialmente, subito dopo l'assegnamento a `x` del valore di `espressione1`.

  Esempi:

  - ```
    for (x, 1, 10)
      write x;
    ```
    Stampa (sulla stessa riga senza spazi intermedi) i numeri da 1 a 10
  - ```
    for (x, 1, x + 10)
      write x;
    ```
    Stampa (sulla stessa riga senza spazi intermedi) i numeri da 1 a 11
  - ```
    for (x, x, 10)
      write x;
    ```
    Stampa (sulla stessa riga senza spazi intermedi) i numeri a partire dal valore contenuto in `x` prima dell'esecuzione del ciclo fino a 10 (non stampa nulla se il valore iniziale di `x` è maggiore di 10).
  - ```
    for (x, x, 2 * x)
      write x;
    ```
    Stampa (sulla stessa riga senza spazi intermedi) i numeri a partire dal valore contenuto in `x` prima dell'esecuzione del ciclo fino al doppio compreso. Se ad esempio `x` contiene inzialmente `5` stampa tutti i numeri da 5 a 10, se `x` contiene inizialmente `0` stampa solo `0`, se `x` contiene inizialmente un numero negativo non stampa nulla.

- Istruzioni di lettura e scrittura:
  L'istruzione `read` seguita dall'identificatore di una variabile intera o da un elemento di un array, legge da input un intero e lo assegna alla variabile o all'elemento indicato.

L'istruzione `write` seguita da un'espressione intera visualizza il risultato dell'espressione (senza portare il cursore a capo).

L'istruzione `writemsg` seguita da una stringa (racchiusa tra virgolette) visualizza la stringa.

L'istruzione `writeln` porta il cursore all'inizio della riga successiva.

- Blocco:
  Istruzioni e dichiarazioni possono essere raggruppate tra parentesi graffe, formando un'*istruzione composta* o blocco.

  Una dichiarazione che si trovi all'interno di un blocco è visibile anche nei blocchi esterni e successivi, a partire dal punto in cui è scritta.

  Si presti attenzione al fatto che le dichiarazioni forniscono informazioni al compilatore e non dipendono in alcun modo dalla sequenza di esecuzione del programma. Pertanto, se la dichiarazione di una variabile si trova nel blocco associato a un ciclo, la variabile è dichiarata una sola volta *indipendentemente* dal numero di ripetizioni del blocco (inoltre essa è dichiarata anche se il blocco non venisse eseguito nel caso di condizione del ciclo immediatamente falsa).

  Analogamente, una dichiarazione che si trovi in uno dei due rami di una selezione, è valida anche nell'altro. Pertanto il seguente frammento di codice non è valido, perché contiene due dichiarazioni del medesimo identificatore:

  ```
  if (...) {
    var x;
    x = 1;
  } else {
    var x;
    x = 2;
  }
  ```

- Istruzione vuota:
  Il punto e virgola chiude le istruzioni (eccetto il blocco che è delimitato dalla parentesi graffa chiusa). Inoltre il punto e virgola da solo costituisce un'istruzione vuota. Ad esempio.

  ```
  for (x, 1, 10000);
  ```

  è un ciclo `for` che esegue diecimila volte l'istruzione che non fa nulla.

Commenti

È possibile introdurre commenti secondo lo stile di C e Java (da `//` a fine riga o tra `/*` e `*/`).

Programma

Un programma è una sequenza di istruzioni e dichiarazioni. Ecco alcuni esempi di programmi[2] [3]:

Esempio 1

```
/* Scarsa fantasia */
writemsg "Hello, world!";
writeln;
```

[2] Come avviene nella maggior parte dei linguaggi di programmazione, l'indentazione e i ritorni a capo non hanno alcuna rilevanza dal punto di vista del compilatore e pertanto possono essere eliminati dall'analizzatore lessicale.

[3] Altri esempi sono disponibili sul sito del corso.

Esempio 2

```
/* Calcolo del massimo comun divisore mediante l'algorimo di Euclide */

var dividendo, divisore;

writemsg "Primo numero? ";
read dividendo;
writemsg "Secondo numero? ";
read divisore;

while (divisore != 0) {
   var resto;
   resto = dividendo % divisore;
   dividendo = divisore;
   divisore = resto;
}
writemsg "il massimo comun divisore e' ";
write dividendo;
writeln;
```

Esempio 3

```
/* Somma di un array di interi */

writemsg "quanti numeri desideri sommare? ";
var quanti;
read quanti;
array numeri[100];

if (quanti <= 100 && quanti >= 0) {
  var i;
  for (i, 0, quanti - 1) {
    writemsg "numero di posizione ";
    write i + 1;
    writemsg "? ";
    read numeri[i];
  }
  var somma;
  somma = 0;
  for (i, 0, quanti - 1)
    somma = somma + numeri[i];

  writemsg "La somma dei numeri letti e' ";
  write somma;
  writeln;
} else {
  writemsg "Posso sommare al massimo 100 numeri!";
  writeln;
}
```

Parte facoltativa: Array dimensionati in esecuzione

Facoltativamente, si può permettere il dimensionamento degli array durante l'esecuzione. In questo caso, la dichiarazione di un array può contenere un'espressione intera. Con questa possibilità nell'Esempio 3, si potrebbe dichiarare l'array utilizzato con

```
array numeri[quanti];
```

dimensionandolo così in base al valore inserito in ingresso.

Nella definizione della grammatica è sufficiente modificare le produzioni della variabile *seqIdentConIndice* sostituendo *espressione* al posto di *numero*. L'implementazione invece cambia notevolmente: nel caso di lunghezza fissata nel testo del programma, il compilatore può decidere quanto (e quale) spazio riservare a ciascun array. Nel caso di lunghezza stabilita in esecuzione, è necessario una gestione dinamica della memoria per gli array.

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.
- Attenzione a non mescolare espressioni di tipo differente. Il linguaggio possiede due tipi: il tipo intero e il tipo array di interi. La sintassi delle espressioni non distingue tra i due tipi. Ad esempio, rispetto alla grammatica fornita è lecito scrivere `x * a`, dove `x` è di tipo intero e `a` è un array di interi. È opportuno implementare un controllo sui tipi che individui queste situazioni anomale.
- È necessario realizzare un controllo sui tipi anche per risolvere l'*overloading* degli operatori. In particolare, l'operatore di assegnamento ha due significati differenti (assegnamento tra interi e assegnamento tra array).

Cosa si richiede

- Scrivere un analizzatore lessicale e un analizzatore sintattico per il linguaggio, servendosi degli strumenti presentati a lezione.
- Scrivere una classe di prova per l'analizzatore lessicale che elenchi i token man mano inseriti.
- Scrivere un compilatore, dal linguaggio sorgente al linguaggio della macchina a stack presentato a lezione. Per generare il codice e per eseguirlo si utilizzino le classi `Codice.java` e `Macchina.java` che NON DEVONO essere modificate.[4]
- Si suggerisce di ispirarsi all'esempio relativo alle espressioni mostrato a lezione. In particolare il parser (generato utilizzando CUP) dovrà costruire una rappresentazione del sorgente mediante un albero, con associata una symbol table. La generazione del codice avverrà a partire da tale albero.
- Non sono richiesti controlli in compilazione e in esecuzione relativamente ai range degli array.
- In caso di errore in compilazione l'applicazione può terminare l'esecuzione, fornendo un breve messaggio relativi all'errore 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:

[4] Anziché generare codice per `Macchina.java` è possibile generare bytecode per la Java Virtual Machine o altre forme di codice intermedio per le quali si disponga di un interprete.

```
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 (nella parte di opzioni e dichiarazioni) il seguente:

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

Si deve consegnare:

1. una descrizione delle classi utilizzate e dell'organizzazione della symbol table;
2. una stampa del file di specifica lessicale e del file di specifica sintattica;
3. una stampa dei sorgenti Java scritti per la risoluzione del problema;
4. una stampa di alcuni esempi di compilazione significativi (sorgente e codice generato);
5. i file sorgenti scritti per la risoluzione del problema, in forma elettronica, con un indice degli stessi.

Il progetto è valido sino a settembre 2009 e deve essere consegnato dieci giorni prima della data concordata per la prova orale (È necesssario iscriversi tramite SIFA all'appello del mese in cui si presenta il progetto).

La grammatica

programma → *seqDichiarEIstr*

seqDichiarEIstr → ϵ
| *seqDichiarEIstr dichiarazione*
| *seqDichiarEIstr istruzione*

dichiarazione → **var** *seqIdentificatori* ;
| **array** *seqIdentConIndice* ;

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

seqIdentConIndice → *identificatore* [*numero*]
| *seqIdentConIndice* , *identificatore* [*numero*]

istruzione → *assegnamento*
| *selezione*
| *cicloWhile*

| *cicloFor*
| *lettura*
| *scrittura*
| *blocco*
| *vuota*

assegnamento → *variabile* `=` *espressione* `;`

selezione → **if** `(` *condizione* `)` *istruzione*
| **if** `(` *condizione* `)` *istruzione* **else** *istruzione*

cicloWhile → **while** `(` *condizione* `)` *istruzione*

cicloFor → **for** `(` *variabile* `,` *espressione* `,` *espressione* `)` *istruzione*

lettura → **read** *variabile* `;`

scrittura → **write** *espressione* `;`
| **writemsg** *stringa* `;`
| **writeln** `;`

blocco → `{` *seqDichiarEIstr* `}`

vuota → `;`

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

condizione → *espressione* `<` *espressione*
| *espressione* `<=` *espressione*
| *espressione* `>` *espressione*
| *espressione* `>=` *espressione*
| *espressione* `==` *espressione*
| *espressione* `!=` *espressione*
| *condizione* `&&` *condizione*
| *condizione* `||` *condizione*
| `!` *condizione*
| `(` *condizione* `)`

variabile → *identificatore*
| *identificatore* `[` *espressione* `]`

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

numero → sequenza non vuota di lettere e cifre

stringa → sequenza di caratteri racchiusa tra virgolette

Si ricordi inoltre che il linguaggio prevede i commenti.