Linguaggi e Traduttori II

Progetto d'esame a.a. 2007/08

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

Variabili e tipi

- Il linguaggio prevede variabili di tipo intero e array di interi.
- Un identificatore di variabile è una sequenza di lettere e cifre che inizia con una lettera.
- Le variabili sono *dichiarate implicitamente* al primo uso. Il tipo di una variabile è desumibile da come essa viene utilizzata (gli identificatori di array sono seguiti da un selettore o appaiono in istruzioni specifiche relative agli array).
- I tipi sono analizzati staticamente durante la fase di *compilazione* del programma, e non dipendono dalle specifiche sequenze di esecuzione.
- Non è possibile utilizzare lo stesso identificatore per una variabile di tipo intero e per un array.

Array

- Le posizioni negli array sono numerate a partire da zero.
- Gli elementi degli array, come in Java o C, sono accessibili mediante un selettore (espressione tra parentesi quadre) di tipo intero.
- Prima di utilizzare un array è necessario dimensionarne la capacità mediante l'istruzione `dim`. Ad esempio, se la variabile intera `x` contiene `5`, l'istruzione

  ```
  dim z[x * 2];
  ```

  assegna all'array `z` capacità `10`. Pertanto, le posizioni accessibili saranno `z[0]`, `z[1]`, ..., `z[9]`. Il tentativo di assegnare capacità negativa a un array deve provocare un errore in esecuzione.
- È possibile assegnare una nuova capacità a un array già esistente. Questo comporta la perdita del contenuto precedente.

Espressioni e condizioni

Il linguaggio prevede le usuali espressioni aritmetiche su interi, costruite a partire da variabili semplici[1] e costanti, 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)`. Vi sono inoltre gli operatori di incremento e decremento prefissi e postifissi, indicati con `++` e `--`, applicabili a variabili o elementi di array, con la stessa semantica prevista dal linguaggio Java.

Un'espressione intera può essere interpretata come *condizione*, secondo quanto previsto dal linguaggio C: un valore diverso da zero indica "vero", mentre un valore uguale a zero indica "falso".

È disponibile l'*operatore condizionale* `?:` analogo a quello di Java:

[1] Una *variabile semplice* è una variabile di tipo intero o un elemento di un array di interi

```
<condizione> ? <espressione1> : <espressione2>
```

Viene valutata la condizione, se è vera si valuta la prima espressione, altrimenti si valuta la seconda. Il risultato prodotto dall'operatore è quello dell'espressione valutata.

Per precedenze e associatività tra gli operatori del linguaggio si faccia riferimento alle regole di Java.

Istruzioni

- Assegnamento:

  ```
  <variabile_semplice> = <espressione>
  ```

- Selezione:

  ```
  if (<condizione>) <istruzione> else <istruzione>
  if (<condizione>) <istruzione>
  ```

- Iterazione:

  ```
  while (<condizione>) <istruzione>
  ```

  L'esecuzione del ciclo termina quando la condizione è falsa.

- Iterazione sugli array:

  ```
  for (<identificatore> : <variabile_array>)
    <istruzione>
  ```

  `<identificatore>` è un nome di variabile, dichiarata implicitamente in questo contesto ed utilizzabile solo all'interno del ciclo; il ciclo viene eseguito assegnando alla variabile via via i valori contenuti nell'array. Ad esempio, alla fine del seguente ciclo la variabile `somma` conterrà la somma dei numeri presenti nell'array riferito da `numeri` (se l'array è vuoto, cioè dimensionato a zero, la variabile conterrà zero; se l'array non è stato dimensionato, dovrà essere fornito un messaggio d'errore e interrotta l'esecuzione):

  ```
  somma = 0;
  for (alfa: numeri)
    somma = somma + alfa;
  ```

- Istruzioni di lettura e scrittura:

  ```
  leggi <variabile_semplice>
  scrivi <variabile_semplice>
  scrivi <costante_stringa>
  acapo
  ```

  La prima istruzione legge da input un intero e lo assegna alla variabile indicata, la seconda scrive in output il valore contenuto nella variabile indicata, la terza scrive in output la `<costante_stringa>` indicata. La costante è una sequenza di caratteri tra virgolette. Le istruzioni `scrivi` lasciano il cursore sulla stessa riga. L'istruzione `acapo` sposta il cursore all'inizio della riga successiva.

- Istruzione composta:
  È una sequenza (anche vuota) di istruzioni, racchiuse tra parentesi graffe aperte e chiuse (come in Java):

```
{
   <istruzione>;
   <istruzione>;
   ...
   <istruzione>;
}
```

Ciascuna istruzione è terminata da un punto e virgola.

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. Ecco alcuni esempi di programmi[2]:

Esempio 1

```
/* Esempio 1: Hello, world! */
scrivi "Hello, world!";
acapo;
```

Esempio 2

```
/* Esempio 2: calcolo del massimo
   comun divisore mediante l'algorimo di
   Euclide
*/
scrivi "dividendo? ";
leggi dividendo;
scrivi "divisore? ";
leggi divisore;

while (divisore) {
   resto = dividendo % divisore;
   dividendo = divisore;
   divisore = resto;
};
scrivi "valore del massimo comun divisore: ";
scrivi dividendo;
acapo;
```

[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.

Esempio 3

```
/* Esempio 3: somma di un array di interi */
scrivi "quanti numeri vuoi sommare? ";
leggi quanti;

dim numeri[quanti];

i = 0;
while (quanti - i) {
  scrivi "numero? ";
  leggi numeri[i++];
};

scrivi "Numeri letti: ";
for (beta: numeri) {
  scrivi beta;
  scrivi " ";
};
acapo;

somma = 0;
for (gamma: numeri)
  somma = somma + gamma;

scrivi "La somma vale ";
scrivi somma;
acapo;
```

Esempio 4

```
/* Esempio 4: il crivello di Eratostene
   (calcolo numeri primi)
*/

scrimsg "Numero massimo da considerare? ";
leggi nMax;

//creazione e inizializzazione array
dim primi[nMax + 2];
x = 2;
while (nMax / x)     //  x <= nMax
  primi[x++] = 1111; //assegna true


numero = 2;
while (nMax / numero) {
  if (primi[numero]) {
    multiplo = numero * 2;
    while (nMax / multiplo) {   // multiplo <= nMax
      primi[multiplo] = 0; //assegna false
      multiplo = multiplo + numero;
    };
```

```
  };
  numero++;
};

scrivi "Elenco primi: ";
numero = 2;
while (nMax / numero) {
  if (primi[numero]) {
    scrivi numero;
    scrivi " ";
  };
  numero++;
};
acapo;
```

Osservazioni

- Le variabili *non* vengono inizializzate automaticamente. Pertanto:
  - una variabile semplice conterrà inizialmente un valore qualunque,
  - il tentativo di accedere a un elemento di un array a cui non sia stata assegnata una capacità provocherà un errore in esecuzione.

  Ad esempio, nel seguente programma la prima e la terza istruzione visualizzano un valore, mentre l'ultima provoca un errore in esecuzione:

  ```
  scrivi x;
  dim y[4];
  scrivi y[2];
  scrivi z[2];
  ```

- Le dichiarazioni sono analizzate staticamente durante la fase di *compilazione* del programma, e non dipendono dalle specifiche sequenze di esecuzione. Pertanto, i due seguenti frammenti di codice devono provocare un errore *in compilazione*:

  ```
  if (x)
    dim z[x * 2];
  else
    z = x;

  if (0)
    y = 1;
  dim y[3];
  ```

  Nel primo frammento si tenta di utilizzare nell'`else` la variabile `z` come fosse di tipo intero, mentre precedentemente è stata utilizzata come array. Nell'ultima istruzione del secondo frammento si tenta di utilizzare `y` come array ma nell'istruzione associata all'`if` la variabile è utilizzata come intera (il fatto che l'istruzione interna all'`if` non venga mai eseguita è irrilevante rispetto all'analisi in compilazione delle dichiarazioni).

- Il dimensionamento degli array avviene in esecuzione. Pertanto, durante l'*esecuzione* dell'ultima istruzione del seguente frammento di codice vi sarà un errore (a meno che in precedenza sia stata eseguita un'istruzione di dimensionamento):

```
if (0)
  dim z[10];
z[1] = 1;
```

Il seguente frammento invece non provoca errori:

```
if (0)
  dim z[5];
else
  dim z[10]
z[9] = z[8];
```

- Gli operatori di incremento prefissi e postfissi hanno una semantica differente. Analogamente quelli di decremento. Alla fine dell'esecuzione dei due frammenti seguenti, il valore di `y` è differente:

```
x = 3;
y = x++;
```

```
x = 3;
y = ++x;
```

- L'operatore condizionale deve valutare *esclusivamente* una delle due espressioni, in base al valore della condizione, e non entrambe. Si noti che nel seguente codice, il risultato visualizzato è differente nel caso si valuti un'espressione e nel caso vengano valutate entrambe:

```
leggi x;
leggi y;
z = x - y ? x++ : y++;
z = x + y + z;
scrivi z;
```

- Attenzione a non rivalutare più volte un'espressione (a fonte di una sola occorrenza nel sorgente). Ad esempio, nel calcolo di `a % b` le espressioni `a` e `b` devono essere valutate una volta sola, sia per motivi di efficienza, sia per motivi di correttezza (infatti potrebbero contenere gli operatori di incremento e decremento che provocano effetti sui contenuti delle variabili).

Parti facoltative

Operatori di confronto

Introdurre gli operatori di confronto `!=`, `==`, `<`, `<=`, ecc. tra interi.

Operatori logici

Introdurre gli operatori logici di congiunzione (`&&`), disgiunzione (`||`) e negazione (`!`), valutati utilizzando la *lazy evaluation*.

Cosa si richiede

- Scrivere una grammatica context–free che specifichi la sintassi del linguaggio.
- 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.[3]
- 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 all'accesso agli array.
- Non è richiesta la garbage collection.
- In caso di errore in compilazione l'applicazione può terminare l'esecuzione (se possibile fornire un messaggio che indichi dove è stato riscontrato l'errore).

Si deve consegnare:

1. una descrizione della grammatica del linguaggio, 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 scritto per la risoluzione del problema, in forma elettronica, con un indice degli stessi.

Il progetto è valido sino a settembre 2008 e deve essere consegnato dieci giorni prima della data concordata per la prova orale[4]*.*

[3] 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.

[4] È necesssario iscriversi tramite SIFA all'appello del mese in cui si presenta il progetto.