Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

19 Giugno 2019

Esercizio 1 (6 punti). Data la grammatica (le lettere minuscole sono simboli terminali)

$$\begin{array}{lcl} S \to & Ab & | \quad Bc \\ A \to & aA & | \quad \varepsilon \\ B \to & acB & | \quad \varepsilon \end{array}$$

Verificare, costruendo l'opportuna tabella, se la grammatica è LL(1). Nel caso non lo sia, esiste un k per cui essa è LL(k). Motivare la risposta.

Esercizio 2 (9 punti). Si assuma di avere un linguaggio con sottotipi (e relazione di sottotipo `<:`).

1. Definire la regola semantica per il comando `x := E` e scrivere in pseudo-codice la funzione `checkStat` che la implementa.

2. Scrivere l'albero di derivazione per il comando

   ```
   x:= y ; y:= z ; z:= new C() ;
   ```

   per l'ambiente $[x \mapsto \mathtt{C}_x, y \mapsto \mathtt{C}_y, z \mapsto \mathtt{C}_z]$. Quela è la relazione tra $\mathtt{C}_x$, $\mathtt{C}_y$ e $\mathtt{C}_z$?

Esercizio 3 (9 punti). Definire la funzione `code_gen` per il comando

```
interleave C and C' upto E times
```

che (1) calcola E e sia v il suo valore e (2) esegue una volta `C` e una volta `C'` in maniera tale che il numero totale di esecuzioni sia v.

Quindi applicare le regole di sopra al comando

```
interleave y := y+1 and x := x-1 upto x+y times
```

assumendo che la variabile `x` si trovi ad offset +4 del frame pointer `$fp`, mentre la variabile `y` si trova nell'ambiente statico immediatamente esterno all'ambiente corrente e a offset +8.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

19 Settembre 2019

Esercizio 1 (6 punti). Definire un analizzatore lessicale in `ANTLR` che accetta sequenze di token che a loro volta sono stringhe **non vuote** sull'alfabeto a, b, c, per cui le occorrenze di a (se ci sono) precedono le occorrenze di b e di c (se ci sono) e le occorrenze di b precedono quelle di c (se ci sono). Ad esempio `a abbc bcc c` è un input riconosciuto.

Esercizio 2 (9 punti). Si assuma di avere un linguaggio object oriented.

1. Definire la regola semantica per l'overriding, ossia la regola che permette di tipare

```
class A { ...
    T1 m (T2 x) { ...}
...}
class B extends A { ...
    T3 m (T4 x) { ...}
...}
```

   [Suggerimento: definire la regola di tipo per il costrutto `class B extends A`, assumendo, per semplicità che `B` abbia esattamente un metodo. Fare i due casi che il metodo sia sovrascritto oppure no]

2. Definire anche la funzione `checkDecs` che implementa la regola del punto precedente.

Esercizio 3 (9 punti). Definire la funzione `code_gen` per il comando

```
whileTre E do { C } { C' } od
```

che (1) calcola `E` e sia v il suo valore; (2) se $v = 0$ allora termina, altrimenti se v è un multiplo di 3 esegue `C`, altrimenti esegue `C'`.

Quindi applicare le regole di sopra al comando

```
whileTre (x+y) { y := y+1; x := x-2 } { x= x-1 } od
```

assumendo che la variabile `x` si trovi ad offset $+4$ del frame pointer `$fp`, mentre la variabile `y` si trova nell'ambiente statico immediatamente esterno all'ambiente corrente e a offset $+8$.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

19 Dicembre 2019

Esercizio 1 (6 punti) Scrivere le definizioni **formali** di `nullable`, `first`, e `follow` per grammatiche LL(1).

Esercizio 2 (10 punti) In un linguaggio di programmazione i programmi `Prg` sono definiti da questa sintassi

$$\begin{array}{lll} \texttt{Prg} & ::= & \texttt{Fun}^* \ \texttt{Stm} \\ \texttt{Fun} & ::= & \texttt{Type Id "(" FPar ")" = Stm} \end{array}$$

dove `Type` possono essere solamente `int` e `bool`, `FPar` sono i parametri formali, cioè sequenze anche vuote del tipo $\texttt{Type}_1\ \texttt{Id}_1, \cdots, \texttt{Type}_n\ \texttt{Id}_n$, e `Stm` è la categoria sintattica dei comandi (lo `Stm` in `Prg` è il *main*). Definire

1. le regole di inferenza per analizzare programmi con mutua ricorsione [Suggerimento: servono due regole, una per costruire l'ambiente iniziale con tutti i tipi delle funzioni, l'altra per analizzare il programma;
2. definire lo pseudocodice per `CheckProg` che implementa le regole di sopra;
3. fornire l'albero di prova per il programma

```
int f(int x) =  return (g(x,x) + 1) ;
int g(int u, int v) =  return(f(u+v)) ;
print(f(1)+g(2,3)) ;
```

   assumendo i vincoli di tipo standard per i comandi e le espressioni (quelli visti a lezione).

Esercizio 3 (8 punti)

1. Definire la funzione `code_gen` per il termine `do S while E` che esegue S, quindi controlla E e se essa è vera riesegue S, altrimenti l'esecuzione termina.
2. Come verifica, si generi il codice di

```
do do ( x:= x+1 ; y:= y+x ) while (x>y) while (y<x+z)
```

   dove le variabili `x`, `y` e `z` si trovano ad offset `+4` e `+8` e `+12` del frame pointer `FP`.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

19 Febbraio 2020

Esercizio 1 (7 punti). Data la grammatica (le lettere minuscole sono simboli terminali, A è il simbolo iniziale)

$$\begin{array}{lll} A \to & BC & \\ B \to & aB & \mid \quad \varepsilon \\ C \to & CbB & \mid \quad c \end{array}$$

Riscrivere la grammatica rimuovendo la ricorsione sinistra e verificare se la grammatica è LL(1) costruendo l'opportuna tabella. Nel caso non lo sia, esiste un k per cui essa è LL(k)? Motivare la risposta.

Esercizio 2 (7 punti). I seguenti sono potenziali regole di tipo per il costrutto `let` in un linguaggio con sottotipaggio (<:). Dire quali regole sono corrette e quali sbagliate. Per quelle sbagliate dare (a) un codice che dovrebbe essere tipabile e non lo è; (b) un codice che è tipabile e invece non dovrebbe essere.

1. $$\frac{\Gamma \vdash e : T' \qquad \Gamma \vdash e' : T'' \qquad T' <: T}{\Gamma \vdash \texttt{let}\ T\ x = e\ \texttt{in}\ e' :\ T''}$$

2. $$\frac{\Gamma \vdash e : T' \qquad \Gamma[x : T] \vdash e' : T'' \qquad T <: T'}{\Gamma \vdash \texttt{let}\ T\ x = e\ \texttt{in}\ e' :\ T''}$$

3. $$\frac{\Gamma \vdash e : T' \qquad \Gamma[x : T'] \vdash e' : T'' \qquad T' <: T}{\Gamma \vdash \texttt{let}\ T\ x = e\ \texttt{in}\ e' :\ T''}$$

Nel caso in cui nessuna regola sia corretta, (i) dare la regola giusta e (ii) controllare che i codici di prima siano correttamente tipabili/non tipabili.

Esercizio 3 (10 punti). Definire la funzione `code_gen` per

1. la dichiarazione di funzione void come: `void f(T1 x, T2 y){ S }` ;
2. l'invocazione di funzione `f(e, e')` (`e`, `e'` sono espressioni).

Quindi, assumendo che l'etichetta che corrisponde alla seguente funzione `fact` sia `fact_label`, scrivere il codice per

```
int x = 1 ;
void fact(int n, int z){
   if (n == 0) x = z ;
   else fact(n-1, z*n) ;
}
```

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

15 Giugno 2020

Nota Bene. Alla fine del compito, fare una foto a tutto il compito col cellulare e inviare le foto per email a `cosimo.laneve@unibo.it` .

Esercizio 1 (6 punti). Definire un analizzatore lessicale in `ANTLR` che accetta sequenze di token che a loro volta sono stringhe non vuote sull?alfabeto $\{\texttt{a}, \texttt{b}\}$ per cui non ci sono mai due occorrenze di `b` consecutive. Ad esempio `a abaa b aaaab` è un input riconosciuto.

Esercizio 2 (7 punti). Data la grammatica (le lettere minuscole sono simboli terminali, A è il simbolo iniziale)

$$\begin{array}{lllllll} S \rightarrow & Aa & | & bAc & |Bc & | & bBa \\ A \rightarrow & aA & | & \varepsilon & & & \\ B \rightarrow & bB & | & \varepsilon & & & \end{array}$$

Verificare se la grammatica è LL(1) costruendo l'opportuna tabella. Nel caso non lo sia, esiste un k per cui essa è LL(k)? Motivare la risposta.

Esercizio 3 (10 punti). Definire la funzione `code_gen` per il comando

```
for id := E to E' do S
```

La semantica del for è: (1) si calcolano il valore delle espressioni `E` e `E'` e siano esse v e v'; (2) quindi si inizializza `id` a v e si esegue `S` se `id` $\leq v'$; (3) dopo l'esecuzione di `S`, si incrementa `id` e si riverifica se `id` $\leq v'$. L'iterazione termina quando `id` $> v'$.

Si applichi tale regola al comando

```
for x := y to z do z := x+1
```

assumendo che le variabili `x` e `y` si trovino nel record di attivazione corrente ad offset 8 e 12 del `$fp`, mentre `z` si trovi nell'ambiente statico immediatamente precedente a offset 8.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

3 Luglio 2020

Nota Bene. Alla fine del compito, fare una foto a tutto il compito col cellulare usando una applicazione che esegue scansioni, tipo CamScanner, e inviarla per email a `cosimo.laneve@unibo.it` .

Si consideri la seguente grammatica (scritta in ANTLR)

```
prg :  'let' dec 'in' stm ;
dec :  (type Id ';')+ ;
type:  'int' | 'double' ;
exp :  Integers | Doubles | Id | exp '+' exp ;
stm :  (Id '=' exp ';')+
```

dove

- gli `Integers` sono sequenze non vuote ndi cifre prefissate dal segno `+` o `-`;
- i `Doubles` sono sequenze non vuote di cifre con esattamente un punto "`.`" e prefissate dal segno `+` o `-`;
- gli `Id` sono gli identificatori (sequenze non vuote di caratteri);
- l'operazione di somma "`+`" è *overloaded*, cioè: in $e1$+$e2$, se sia $e1$ che $e2$ sono interi, allora il risultato è un intero, altrimenti è un double;
- nell'assegnamento `x = e ;`
  - se `x` è intero ed `e` è double allora il valore di `e` viene troncato prima di essere memorizzato in `x`;
  - se `x` è double ed `e` è intero allora il valore di `e` viene esteso con "`.0`" prima di essere memorizzato in `x`.

Esercizi

9 1. dare tutte le regole di inferenza per la verifica dei tipi del linguaggio di sopra.
[**SUGGERIMENTO**: La regola di inferenza del programma ritorna un `stm` in un linguaggio esteso in cui si aggiungono i cast espliciti "`x = (double)e ;`" oppure "`x = (int)e ;`" dove sono necessari;]

4 2. verificare, scrivendo l'albero di prova, che il programma seguente sia correttamente tipato:

```
let double x; int y; in y = 5.4 ; x = 3 + y ;
```

2 3. scrivere un programma che non sia tipabile nel sistema definito e spiegarne il motivo;

9 4. definire il codice intermedio di `e1 + e2`, di `x = e ;` (e, nel caso si siano aggiunti i cast espliciti, di `x = (double)e ;` di `x = (int)e ;`) assumendo che

(a) tutti i registri sono a 8 byte (memorizzano double);

(b) ci siano due operazioni di addizione: `iadd $r1 $r2 $r3` e `dadd $r1 $r2 $r3`. L'operazione `iadd $r1 $r2 $r3` fa la somma prendendo la parte intera di `$r1` ed `$r2` e memorizzano il risultato in `$r3` (con un sufisso "`.0`"); `dadd` fa la somma tra double.

(c) c'è un'operazione `isw $r0 k($r1)` che memorizza la parte intera di `$r0` ad offset `k` dell'indirizzo in `$r1`. In questo caso tale indirizzo occupa 4 byte.

(d) c'è un'operazione standard `sw $r0 k($r1)` che memorizza `$r0` ad offset `k` dell'indirizzo in `$r1`. In questo caso tale indirizzo occupa 8 byte.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

20 Luglio 2020

Nota Bene. Quando avete terminato, fare una foto a tutto il compito col cellulare usando una applicazione che esegue scansioni, tipo CamScanner, e inviarla per email a `cosimo.laneve@unibo.it` .

Si consideri la seguente grammatica (scritta in ANTLR)

```
prg :  'let' dec 'in' stm ;
dec :  ('int' Id ';')+ ;
exp :  Integers | Id | exp '+' exp ;
stm :  (Id '=' exp ';')+
```

dove

- gli `Integers` sono sequenze non vuote di cifre prefissate dal segno `+` o `-`;
- gli `Id` sono gli identificatori (sequenze non vuote di caratteri);

Esercizi

1. (**punti 2**) completare l'input di ANTLR con le regole per l'analizzatore lessicale che riguardano `Integers` e `Id`;
2. (**punti 9**) dare tutte le regole di inferenza per verificare l'uso di identificatori non inizializzati. Ad esempio `let int x; int y; in x = 3 + y ;` è un programma erroneo secondo l'analisi semantica. L'analisi semantica ritorna anche informazioni sull'offset degli identificatori (vedi punto 4);
3. (**punti 4**) verificare, scrivendo l'albero di prova, che il programma seguente sia correttamente tipato:

   `let int x; int y; in y = 5 ; x = 3 + y ;`

4. (**punti 9**) definire il codice intermedio *per tutti i costrutti del linguaggio*, in particolare allocando lo spazio necessario sulla pila per memorizzare i valori degli identificatori (che occupano sempre 4 byte).

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

18 Settembre 2020

Nota Bene. Quando avete terminato, fare una foto a tutto il compito col cellulare usando una applicazione che esegue scansioni, tipo CamScanner, e inviarla per email a `cosimo.laneve@unibo.it` .

Esercizio 1 (punti 6) Gli identificatori di un linguaggio di programmazione devono iniziare e terminare con "_" e tra questi due caratteri ci possono essere solo lettere maiuscole e cifre (in qualunque ordine) con il vincolo che il numero di lettere e quello delle cifre sia sempre pari. Definire l'analizzatore lessicale per questi identificatori in `ANTLR`.

Esercizio 2 (punti 9) Si consideri la seguente grammatica:

```
S  ->  S B   |   y
B  ->  B x   |   A x
A  ->  z     |   z S y
```

1. verificare, costruendo la tabella che la grammatica non è LL(1);
2. modificare la grammatica per renderla LL(1) e dimostrarlo costruendo la tabella.

Esercizio 3 (punti 9) Si consideri il comando iterativo

$$\texttt{loop}\ k\ \{S\}$$

dove k è una costante intera. Quando $k > 0$, questo comando itera esattamente k volte il corpo S. Quando $k \leq 0$ il comando non fa niente.

1. Scrivere la `cgen` per questo comando;
2. generare il codice intermedio per

   ```
   loop 34 { x = x+1 ; loop 25 { y = x+y ;} }
   ```

   assumendo che `x` sia ad offset 4 del record di attivazione corrente e `y` sia ad offset 4 del record di attivazione dell'ambiente statico immediatamente esterno.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

19 Febbraio 2021

Esercizio 1 (7 punti) Sia L il linguaggio sull'alfabeto $\{a, b, c, d\}$ costituito da stringhe della forma $\alpha d \beta$ dove α è una qualunque stringa non vuota che contiene $\{b, c\}$ e β è una qualunque stringa non vuota che contiene $\{a, c\}$. Si definisca in `ANTLR` l'analizzatore lessicale per tokens in L.

Esercizio 2 (7 punti) Si verifichi che la grammatica

$$\begin{array}{lcl} S & \rightarrow & AaB \mid B \\ A & \rightarrow & bB \mid c \mid \varepsilon \\ B & \rightarrow & aA \end{array}$$

(ε è la stringa vuota). Verificare, scrivendo la tabella relativa, che

- la grammatica è LL(1) ;

Esercizio 3 (1) Definire la funzione `code_gen` che prende in input un termine

$$E \texttt{ \&\& } E'$$

e genera il codice intermedio (una espressione booleana ritorna 0, per falso, o 1, per vero). Il valore di ritorno si trova, come per tutte le espressioni, nel registro `$a0`.

(2) Come verifica, scrivere il codice di

```
(x && y) && z
```

Assumendo che le variabili `x`, `y` e `z` si trovano ad offset `+4`, `+8` del frame pointer `$fp`, mentre la variabile `z` si trova nell'ambiente statico immediatamente esterno all'ambiente corrente e a offset 0 (l'ambiente statico è accessibile attraverso il registro `$a1`).

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

28 Maggio 2021

Nota Bene. Quando avete terminato, fare una foto a tutto il compito col cellulare usando una applicazione che esegue scansioni, tipo CamScanner, e inviarla per email a `cosimo.laneve@unibo.it` .

I programmi di un linguaggio di programmazione sono blocchi `Dec Stm` dove

- `Dec` sono sequenze di dichiarazioni di identificatori interi (int);
- `Stm` sono sequenze di comandi che possono essere
  - assegnamenti di una espressione `Exp` a una variabile;
  - iterazioni while (la guardia del condizionale è una espressione intera, la semantica è quella di `C`).
- `Exp` possono essere costanti intere, identificatori o espressioni con somma.

Esercizi

1. (**punti 6**) definire l'input *completo* di `ANTLR` per la grammatica del linguaggio di sopra;
2. (**punti 9**) dare tutte le regole di inferenza per verificare il corretto uso degli identificatori (identificatori non dichiarati o di dichiarazioni multiple) e per gestire gli offset nella generazione di codice.
3. (**punti 9**) definire il codice intermedio *per tutti i costrutti del linguaggio*, in particolare per il programma. Ricordate che la `cgen` prende come input anche l'ambiente/tabella dei simboli nei vari nodi dell'albero sintattico. Fate attenzione alla gestione degli accessi al record di attivazione.

   Generare il codice intermedio per il codice

```
int x; int z;
x = 4; z = x+5; while (z + 3){ z = z+x ; while (x){ x = x+1; } }
```

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

16 Giugno 2021

Nota Bene. Quando avete terminato, fare una foto a tutto il compito col cellulare usando una applicazione che esegue scansioni, tipo CamScanner, e inviarla per email a `cosimo.laneve@unibo.it` .

I programmi di un linguaggio di programmazione sono blocchi { `Dec Stm` } dove

- `Dec` sono sequenze di dichiarazioni di identificatori interi;
- `Stm` sono sequenze di comandi che possono essere
  - assegnamenti;
  - condizionali (la guardia del condizionale ha tipo intero, la semantica è quella di `C`);
  - blocchi { `Dec Stm` }. Attenzione: i blocchi possono essere annidati.
- `Exp` possono essere naturali, identificatori o espressioni con somma.

Esercizi

1. (**punti 6**) definire l'input *completo* di `ANTLR` per la grammatica del linguaggio di sopra;
2. (**punti 9**) dare tutte le regole di inferenza per verificare il corretto uso degli identificatori (identificatori non dichiarati o di dichiarazioni multiple) e per gestire gli offset nella generazione di codice.
3. (**punti 9**) definire il codice intermedio *per tutti i costrutti del linguaggio*, in particolare allocando lo spazio necessario sulla pila per gestire i blocchi. Ricordate che la `cgen` prende come input anche l'ambiente/tabella dei simboli nei vari nodi dell'albero sintattico. Fate attenzione alla gestione degli accessi ai record di attivazione. Osservate anche che la grandezza massima del record di attivazione si può stabilire staticamente.

   Generare il codice intermedio per il codice

   ```
   { int x; x = 1; if (x+1) { int z; z = x+5; } else {int y; y = x+1; }
   ```

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

21 Giugno 2022

Si consideri il linguaggio di programmazione la cui sintassi in `ANTLR` è la seguente:

```
prog : '{' (dec)? (stm)+ '}' ;
dec  : 'int' (ID ',')* ID ';';
stm  : ID '=' exp ';' | 'if' '(' exp ')' stm 'else' stm | block  ;
block: '{' (dec)? (stm)+ '}' ;
exp  : ID | NUM | exp ('+' | '-') exp ;
ID   : ('a'..'z')+ ;
NUM  : ('0'..'9')+ ;
```

dove la guardia del condizionale è vera quando `exp` è diverso da 0, falsa altrimenti.

Un identificatore `ID` è considerato "*costante*" quando viene assegnato una sola volta (si trova una sola volta come left-hand side expression di un assegnamento) e gli eventuali identificatori della right-hand side expression sono anch'essi costanti. [Immaginare che se un identificatore è costante nel tempo, allora è possibile rimuoverlo rimpiazzandolo con il valore costante.]

Esercizi

1. (**punti 9**) definire tutte le regole di inferenza per verificare gli identificatori costanti e per gestire gli offset nella generazione di codice. In particolare
   - definire il dominio astratto e le operazioni su di esso da usare nelle regole semantiche e dimostrare che le operazioni sono monotone;
   - per quanto riguarda gli offset, la memoria per eseguire il codice dovrà essere tutta in un unico frame (non si dovranno usare liste di frames);
2. (**punti 6**) scrivere gli alberi di prova per i seguenti programmi:

   ```
   { int x, y, z ; x = 1; y = 2 ; if (x + y) { z= x+y ;} else { z = 4 ;} x = z+y ; }
   ```

   ```
   { int x, y, z ; x = 1; y = 2 ; if (x - x) { z= x+y ;} else { z = 3 ;} x = x+1 ; }
   ```

   ```
   { int x, y ; if (x + y) { x = 1; y = 2 ; } else { x = y +1 ; } }
   ```

3. (**punti 9**) definire il codice intermedio *per tutti i costrutti del linguaggio*, in particolare tenendo conto del vincolo che non ci dovranno essere più frames.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

9 Luglio 2021

Nota Bene. Quando avete terminato, fare una foto a tutto il compito col cellulare usando una applicazione che esegue scansioni, tipo CamScanner, e inviarla per email a `cosimo.laneve@unibo.it` .

I programmi di un linguaggio di programmazione sono blocchi { `Dec Stm` } dove

- `Dec` sono sequenze di dichiarazioni di identificatori interi con inizializzazioni, cioè `Dec : ('int' X '=' Exp ';')* ;`
- `Stm` sono sequenze di comandi. Cioè

```
Stm : ( X '=' Exp ';' | 'if' '(' Exp ')' '{' Stm '}' 'else' '{' Stm '}' )+ ;
```

- la sintassi delle espressioni `Exp` è:

$$\texttt{Exp}: \quad \texttt{Exp '+' Exp} \;\mid\; \texttt{Exp '-' Exp} \;\mid\; \texttt{X} \;\mid\; \texttt{N};$$

dove `N` sono i naturali, e `X` sono gli identificatori.

Esercizi

1. (**punti 6**) trasformare la grammatica delle espressioni in modo da eliminare la ricorsione sinistra. Quindi verificare, costruendo l'opportuna tabella, che la grammatica ottenuta sia LL(1). [Assumere che `N` ed `X` siano simboli terminali.]

2. (**punti 9**) dare tutte le regole di inferenza per verificare se un identificatore contiene un numero pari o un numero dispari, il corretto uso degli identificatori (identificatori non dichiarati o di dichiarazioni multiple) e per gestire gli offset nella generazione di codice. Per quanto riguarda, la verifica pari/dispari, assumere di avere un test sulle costanti `Is_oe(n)` che ritorna `p` se `n` è pari, `d` se è dispari. Inoltre si ricordi che la somma/differenza di due pari o di due dispari è pari, mentre la somma/differenza di un pari e un dispari è dispari. Quando non si è in grado di determinare se un identificatore è pari o dispari allora quell' identificatore ha valore $\top$ (quando uno degli argomenti della somma è $\top$ allora il risultato è $\top$. Fare attenzione al comando condizionale.

   Scrivere l'albero di prova per

```
{ int x = 1; int z = x + 3; if (x+1) { z = x+z; } else {z = x+1; x = z-1 ;}
```

3. (**punti 9**) definire il codice intermedio *per tutti i costrutti del linguaggio*, in particolare allocando lo spazio necessario sulla pila per gestire i blocchi.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

17 Settembre 2021

Nota Bene. Alla fine del compito, fare una foto a tutto il compito col cellulare usando una applicazione che esegue scansioni, tipo CamScanner, e inviarla per email a `cosimo.laneve@unibo.it` .

Si consideri la seguente grammatica (scritta in ANTLR)

```
prg : stm ;
stm : (Id '=' exp ';' | '{' dec stm '}')+ ;
dec : (type Id ';')+ ;
type: 'int' | 'bool' ;
exp : Int | Bool | Id | exp '+' exp | exp '-' exp | exp '>' exp | exp '==' exp
    | exp '||' exp | exp '&&' exp ;
```

dove

- gli `Int` sono sequenze non vuote di cifre senza segno oppure prefissate dal segno `+` o `-`;
- i `Bool` sono i valori "`true`" e "`false`";
- gli `Id` sono gli identificatori (sequenze non vuote di caratteri);
- le operazioni di somma "`+`", sottrazione "`-`" e maggiore "`>`" si applicano a espressioni `Int`, le operazioni di or "`||`" e and "`&&`" si applicano a espressioni `Bool`, mentre l'operazione di uguaglianza si applica a espressioni dello stesso tipo.

Esercizi

1. dare tutte le regole di inferenza per la verifica dei tipi del linguaggio di sopra (attenzione che il linguaggio ammette annidamento di ambienti);
2. verificare, scrivendo l'albero di prova, che il programma seguente sia correttamente tipato:

   `{ int x; int y; x = 5; { bool z; z = (x > 5)||false; { int z; y = 6+x; z = 3 + y; }}}`
3. definire il codice intermedio per tutti i costrutti del linguaggio. In particolare, si ricordi che un booleano occupa 1 byte mentre un intero occupa 4 byte. Inoltre, per quanto riguarda le operazioni `||` e `&&`, si implementi la cosiddetta *lazy evaluation*: il secondo argomento non viene valutato se inutile (nell' `||`, il secondo argomento non viene valutato se il primo è `true`, nell' `&&` il secondo argomento non viene valutato se il primo è `false`).
4. scrivere il codice generato per l'esempio al punto 2.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

27 Maggio 2022

I programmi di un linguaggio di programmazione sono blocchi { `Dec Stm` } dove

- `Dec` sono sequenze di dichiarazioni di identificatori interi o booleani;
- `Stm` sono sequenze di comandi che possono essere
  - assegnamenti;
  - condizionali if-then-else ;
  - blocchi { `Dec Stm` }. Attenzione: i blocchi possono essere annidati.
- `Exp` possono essere naturali, booleani, identificatori, espressioni con somma o congiunzioni (`&&`) di espressioni.

Esercizi

1. (**punti 6**) definire l'input *completo* di `ANTLR` per la grammatica del linguaggio di sopra (incluse le escape sequences);
2. (**punti 9**) dare tutte le regole di inferenza per verificare il corretto uso degli identificatori (identificatori non dichiarati, dichiarazioni multiple o errori di tipo) e per gestire gli offset nella generazione di codice.
3. (**punti 9**) definire il codice intermedio *per tutti i costrutti del linguaggio*, in particolare facendo attenzione alla gestione dei blocchi. Ricordate che la `cgen` prende come input anche l'ambiente/tabella dei simboli nei vari nodi dell'albero sintattico.

   Generare il codice intermedio per il codice

```
{ int x; bool y; x = 1; y = true; if (y && false) { int y; y = x+5; } else y = false; }
```

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

21 Giugno 2022

Si consideri il linguaggio di programmazione la cui sintassi in ANTLR è la seguente:

```
prog : '{' (dec)? (stm)+ '}' ;
dec  : 'int' (ID ',')* ID ';';
stm  : ID '=' exp ';' | 'if' '(' exp ')' stm 'else' stm | block  ;
block: '{' (dec)? (stm)+ '}' ;
exp  : ID | NUM | exp ('+' | '-') exp ;
ID   : ('a'..'z')+ ;
NUM  : ('0'..'9')+ ;
```

dove la guardia del condizionale è vera quando exp è diverso da 0, falsa altrimenti.

Un identificatore ID è considerato "*costante*" quando viene assegnato una sola volta (si trova una sola volta come left-hand side expression di un assegnamento) e gli eventuali identificatori della right-hand side expression sono anch'essi costanti. [Immaginare che se un identificatore è costante nel tempo, allora è possibile rimuoverlo rimpiazzandolo con il valore costante.]

Esercizi

1. (**punti 9**) definire tutte le regole di inferenza per verificare gli identificatori costanti e per gestire gli offset nella generazione di codice. In particolare
   - definire il dominio astratto e le operazioni su di esso da usare nelle regole semantiche e dimostrare che le operazioni sono monotone;
   - per quanto riguarda gli offset, la memoria per eseguire il codice dovrà essere tutta in un unico frame (non si dovranno usare liste di frames);
2. (**punti 6**) scrivere gli alberi di prova per i seguenti programmi:

```
{ int x, y, z ; x = 1; y = 2 ; if (x + y) { z= x+y ;} else { z = 4 ;} x = z+y ; }

{ int x, y, z ; x = 1; y = 2 ; if (x - x) { z= x+y ;} else { z = 3 ;} x = x+1 ; }

{ int x, y ; if (x + y) { x = 1; y = 2 ; } else { x = y +1 ; } }
```

3. (**punti 9**) definire il codice intermedio *per tutti i costrutti del linguaggio*, in particolare tenendo conto del vincolo che non ci dovranno essere più frames.

Corso di Laurea Magistrale in Informatica

Compito di Compilatori e Interpreti

13 Luglio 2022

Esercizio 1 (6 punti) Scrivere le definizioni **formali** di `nullable`, `first`, e `follow` per grammatiche *LL(1)*.

Esercizio 2 (10 punti) In un linguaggio di programmazione i programmi `prg` sono definiti da questa sintassi ANTLR

```
prg  :  dec* exp ;
dec  :  type ID '=' exp ';' | type ID '(' fPar ')' '=' exp ';' ;
fPar :  type ID (',' type ID)* ;
exp  :  NUM | ID | ID '(' exp (',' exp)* ')' | exp '+' exp | exp '&&' exp ;
```

dove `type` possono essere solamente `int` e `bool`. Definire

1. le regole di inferenza per analizzare programmi con mutua ricorsione [Suggerimento: servono due regole, una per costruire l'ambiente iniziale con tutti i tipi delle funzioni, l'altra per analizzare il programma;
2. definire lo pseudocodice per `CheckProg` che implementa le regole di sopra;
3. fornire l'albero di prova per il programma

```
int z = 3 ;
int f(int x) = g(x,z) + 1 ;
int g(int u, int v) = f(u+v) ;
f(1)+g(2,z)
```

~~assumendo bisogni di tipo standard per i comandi e le espressioni (quelli visti a lezione).~~

Esercizio 3 (8 punti)

1. Definire la funzione `code_gen` per il termine `do S while E` che esegue S, quindi controlla E e se essa è vera riesegue S, altrimenti l'esecuzione termina.
2. Come verifica, si generi il codice di

```
do do ( x:= x+1 ; y:= y+x ) while (x>y) while (y<x+z)
```

dove le variabili `x`, `y` e `z` si trovano ad offset `+4` e `+8` e `+12` del frame pointer `FP`.