

Iteratore **while-do** e comando **continue** del C in Small21

Cecilia Marchi

Università di Pisa

29 Ottobre 2021

- **while-do**

- É un costrutto di iterazione con sintassi `while E do C`.
- L'iterazione è non determinata, ovvero il numero di iterazioni non può essere determinato utilizzando lo stato iniziale e l'espressione di controllo `E`.
- L'iterazione può anche non terminare.

- **continue**

- Il comando può trovarsi solo nel blocco iterato.
- É un controllo di sequenza esplicito.
- Conclude la corrente iterazione ed avvia la rivalutazione dell'espressione di controllo.

- **while-do**

- È un costrutto di iterazione con sintassi `while E do C`.
- L'iterazione è non determinata, ovvero il numero di iterazioni non può essere determinato utilizzando lo stato iniziale e l'espressione di controllo `E`.
- L'iterazione può anche non terminare.

- **continue**

- Il comando può trovarsi solo nel blocco iterato.
- È un controllo di sequenza esplicito.
- Conclude la corrente iterazione ed avvia la rivalutazione dell'espressione di controllo.

Il costrutto **while-do** deve essere utilizzabile in ogni punto del programma. Il suo comportamento può essere emulato con il comando **goto**, ma solo nel blocco più esterno.

Inoltre Small21 non contiene costrutti iterativi.

Per questo, l'aggiunta di **while-do** nel linguaggio ne aumenterà notevolmente l'espressività.

Esempi di programmi che vorremmo scrivere con i nuovi comandi.

```
Program es1 {
  int x = 0, y = 1;
  int temp = 0;
  while (y < 100) {
    temp = x+y;
    x = y;
    y = temp;
  }
}
```

```
Program es2 {
  int x = 0;
  while (x < 3) {
    if (x == 1) {
      x = x + 2;
      continue;
    }
    x = x + 1;
  }
}
```

```
Program es3 {
  int x = 10;
  int y = 0;
  while (x > 7) do {
    while (y < 3) do {
      if (y == 2) {
        y = y + 2;
        continue;
      }
      y = y + 1;
    }
    x = x - 1;
  }
}
```

Esempi di programmi che vorremmo scrivere con i nuovi comandi.

```
Program es1 {
  int x = 0, y = 1;
  int temp = 0;
  while (y < 100) {
    temp = x+y;
    x = y;
    y = temp;
  }
}
```

```
Program es2 {
  int x = 0;
  while (x < 3) {
    if (x == 1) {
      x = x + 2;
      continue;
    }
    x = x + 1;
  }
}
```

```
Program es3 {
  int x = 10;
  int y = 0;
  while (x > 7) do {
    while (y < 3) do {
      if (y == 2) {
        y = y + 2;
        continue;
      }
      y = y + 1;
    }
    x = x - 1;
  }
}
```

Esempi di programmi che vorremmo scrivere con i nuovi comandi.

```
Program es1 {
  int x = 0, y = 1;
  int temp = 0;
  while (y < 100) {
    temp = x+y;
    x = y;
    y = temp;
  }
}
```

```
Program es2 {
  int x = 0;
  while (x < 3) {
    if (x == 1) {
      x = x + 2;
      continue;
    }
    x = x + 1;
  }
}
```

```
Program es3 {
  int x = 10;
  int y = 0;
  while (x > 7) do {
    while (y < 3) do {
      if (y == 2) {
        y = y + 2;
        continue;
      }
      y = y + 1;
    }
    x = x - 1;
  }
}
```

- **Sintassi concreta: una CFG per Small21**

`Stm` \rightarrow `...` | `while (Exp) do Stm`

`OtherStm` \rightarrow `...` | `continue`

- **Modifiche Sintassi Astratta di Small21**

`Stm ::=...` | `[whileDo] Exp Stm` | `[continue]`

Occorre anche aggiungere un nuovo header `[wD]` per gli AR dello stack di Small21.

- **Sintassi concreta: una CFG per Small21**

$Stm \rightarrow \dots \mid \text{while (Exp) do } Stm$

$OtherStm \rightarrow \dots \mid \text{continue}$

- **Modifiche Sintassi Astratta di Small21**

$Stm ::= \dots \mid [\text{whileDo}] \text{ Exp } Stm \mid [\text{continue}]$

Occorre anche aggiungere un nuovo header [wD] per gli AR dello stack di Small21.

- **Sintassi concreta: una CFG per Small21**

$Stm \rightarrow \dots \mid \text{while (Exp) do Stm}$

$OtherStm \rightarrow \dots \mid \text{continue}$

- **Modifiche Sintassi Astratta di Small21**

$Stm ::= \dots \mid [\text{whileDo}] \text{Exp Stm} \mid [\text{continue}]$

Occorre anche aggiungere un nuovo header [WD] per gli AR dello stack di Small21.

$$[Y1'] \frac{\langle e, Y_\rho \rangle \rightarrow_Y ([\text{bool}], Y_\rho) \quad \langle \text{stm}, Y_\rho \rangle \rightarrow_Y ([\text{void}], Y'_\rho)}{\langle [\text{whileDo}] e \text{ stm}, Y_\rho \rangle \rightarrow_Y ([\text{void}], Y_\rho)}$$

$$[Y2'] \frac{}{\langle [\text{continue}], Y_\rho \rangle \rightarrow_Y ([\text{void}], Y_\rho)}$$

$$[Y1'] \frac{\langle e, Y_\rho \rangle \rightarrow_{\mathcal{Y}} ([\text{bool}], Y_\rho) \quad \langle \text{stm}, Y_\rho \rangle \rightarrow_{\mathcal{Y}} ([\text{void}], Y'_\rho)}{\langle [\text{whileDo}] e \text{ stm}, Y_\rho \rangle \rightarrow_{\mathcal{Y}} ([\text{void}], Y_\rho)}$$

$$[Y2'] \frac{}{\langle [\text{continue}], Y_\rho \rangle \rightarrow_{\mathcal{Y}} ([\text{void}], Y_\rho)}$$

$$[E1'.1] \frac{\langle e, Y_\rho \rangle \rightarrow_Y (t, Y_\rho) \quad t \neq [\text{bool}]}{\langle [\text{whileDo}] e \text{ stm}, Y_\rho \rangle \rightarrow_Y ([\text{terr}], Y_\rho)}$$

$$[E1'.2] \frac{\langle e, Y_\rho \rangle \rightarrow_Y ([\text{bool}], Y_\rho) \quad \langle \text{stm}, Y_\rho \rangle \rightarrow_Y ([\text{terr}], Y'_\rho)}{\langle [\text{whileDo}] e \text{ stm}, Y_\rho \rangle \rightarrow_Y ([\text{terr}], Y_\rho)}$$

$$[E1'.1] \frac{\langle e, Y_\rho \rangle \rightarrow_Y (t, Y_\rho) \quad t \neq [\text{bool}]}{\langle [\text{whileDo}] e \text{ stm}, Y_\rho \rangle \rightarrow_Y ([\text{terr}], Y_\rho)}$$

$$[E1'.2] \frac{\langle e, Y_\rho \rangle \rightarrow_Y ([\text{bool}], Y_\rho) \quad \langle \text{stm}, Y_\rho \rangle \rightarrow_Y ([\text{terr}], Y'_\rho)}{\langle [\text{whileDo}] e \text{ stm}, Y_\rho \rangle \rightarrow_Y ([\text{terr}], Y_\rho)}$$

$$\begin{aligned}
 & \sigma = (\text{ar} + \Delta, \mu) \\
 & \langle \mathbf{e}, \sigma \rangle \rightarrow [[\text{bool}], \text{true}, \sigma_e] \\
 & \quad \sigma_e = (\Delta_e, \mu_e) \\
 & \quad \{[\text{WD}], 1, [], \mathbf{s}, []\} = \text{ar}_{\text{top}} \\
 & \quad (\text{ar}_{\text{top}} + \Delta_e, \mu_e) \rightarrow_{R^*} (\text{ar}' + \Delta', \mu') \\
 & \quad \Delta' = \text{ar}_1 + \Delta'' \quad \text{ar}_1 = \{\mathbf{h}_1, \mathbf{l}_1, \mathbf{f}_1, \mathbf{c}_1, \mathbf{r}_1\} \\
 & \quad \quad \{\mathbf{h}_1, \mathbf{l}_1, \mathbf{f}_1, \mathbf{c} + \mathbf{c}_1, \mathbf{r}_1\} = \text{ar}'_1 \\
 & \quad \quad \mathbf{c} = [\text{whileDo}] \ \mathbf{e} \ \mathbf{s} \\
 & \quad \quad (\text{ar}'_1 + \Delta'', \mu') = \sigma_F \\
 [S1'] \frac{}{\langle [\text{whileDo}] \ \mathbf{e} \ \mathbf{s}, \sigma \rangle \rightarrow ([\text{void}], \sigma_F)}
 \end{aligned}$$

$$[S2'] \frac{\langle \mathbf{e}, \sigma \rangle \rightarrow [[\text{bool}], \text{false}, \sigma_e]}{\langle [\text{whileDo}] \ \mathbf{e} \ \mathbf{s}, \sigma \rangle \rightarrow ([\text{void}], \sigma_e)}$$

$$\begin{aligned}
 & \sigma = (\text{ar} + \Delta, \mu) \\
 & \langle \mathbf{e}, \sigma \rangle \rightarrow \llbracket [\text{bool}], \text{true}, \sigma_e \rrbracket \\
 & \sigma_e = (\Delta_e, \mu_e) \\
 & \{[\text{WD}], 1, [], \mathbf{s}, []\} = \text{ar}_{\text{top}} \\
 & (\text{ar}_{\text{top}} + \Delta_e, \mu_e) \rightarrow_{R^*} (\text{ar}' + \Delta', \mu') \\
 & \Delta' = \text{ar}_1 + \Delta'' \quad \text{ar}_1 = \{\mathbf{h}_1, \mathbf{l}_1, \mathbf{f}_1, \mathbf{c}_1, \mathbf{r}_1\} \\
 & \{\mathbf{h}_1, \mathbf{l}_1, \mathbf{f}_1, \mathbf{c} + \mathbf{c}_1, \mathbf{r}_1\} = \text{ar}'_1 \\
 & \mathbf{c} = [\text{whileDo}] \ \mathbf{e} \ \mathbf{s} \\
 & (\text{ar}'_1 + \Delta'', \mu') = \sigma_F \\
 [S1'] \frac{}{\langle [\text{whileDo}] \ \mathbf{e} \ \mathbf{s}, \sigma \rangle \rightarrow ([\text{void}], \sigma_F)} \\
 [S2'] \frac{\langle \mathbf{e}, \sigma \rangle \rightarrow \llbracket [\text{bool}], \text{false}, \sigma_e \rrbracket}{\langle [\text{whileDo}] \ \mathbf{e} \ \mathbf{s}, \sigma \rangle \rightarrow ([\text{void}], \sigma_e)}
 \end{aligned}$$

$$\begin{aligned}
 & \sigma = (ar + \Delta, \mu) \\
 & ar = \{h, l, f, c, r\} \quad h \neq [WD] \\
 & \Delta = ar_1 + \Delta_1 \\
 & ar_1 = \{h_1, l_1, f_1, c_1, r_1\} \\
 & [continue] = s \\
 & \{h_1, l_1, f_1, s, r_1\} = ar'_1 \\
 & (ar'_1 + \Delta_1, \mu) = \sigma_F \\
 [S3'.1] & \frac{}{\langle [continue], \sigma \rangle \rightarrow ([void], \sigma_F)}
 \end{aligned}$$

$$\begin{aligned}
 & \sigma = (ar + \Delta, \mu) \quad \Delta = ar_1 + \Delta_1 \\
 & ar = \{[WD], l, f, c, r\} \\
 [S3'.2] & \frac{}{\langle [continue], \sigma \rangle \rightarrow ([void], (\Delta, \mu))}
 \end{aligned}$$

$$\begin{aligned}
 & \sigma = (ar + \Delta, \mu) \\
 & ar = \{h, l, f, c, r\} \quad h \neq [WD] \\
 & \Delta = ar_1 + \Delta_1 \\
 & ar_1 = \{h_1, l_1, f_1, c_1, r_1\} \\
 & [continue] = s \\
 & \{h_1, l_1, f_1, s, r_1\} = ar'_1 \\
 & (ar'_1 + \Delta_1, \mu) = \sigma_F \\
 [S3'.1] & \frac{}{\langle [continue], \sigma \rangle \rightarrow ([void], \sigma_F)}
 \end{aligned}$$

$$\begin{aligned}
 & \sigma = (ar + \Delta, \mu) \quad \Delta = ar_1 + \Delta_1 \\
 & ar = \{[WD], l, f, c, r\} \\
 [S3'.2] & \frac{}{\langle [continue], \sigma \rangle \rightarrow ([void], (\Delta, \mu))}
 \end{aligned}$$

Modifiche all'interprete

```
stm =  
  Upd of exp * exp  
  | IfT of exp * stm  
  | Goto of lab  
  | SeqS of stm * stm  
  | BlockS of dcl * stm  
  | Call of ide * apars  
  | Return of exp  
  | WhileDo of exp * stm  
  | Continue  
  | ES  
  and  
  
(* =====  
(* Activation Records *)  
(* =====  
  
type head = Name of ide | Exp | Cmd | NoneH | IPB | WD;;  
type stch = int;;  
type frame = env;;
```

Modifiche all'interprete

```
toStringStm tab = function
| WhileDo(exp,stm) -> (let expString =
                        (if isAtomic exp then "(" ^ (toStringExp exp) ^ ")")
                        else toStringExp exp)
                        and stmString = toStringStm (tab + 1) stm
                        in
                        (indent tab) ^ "while " ^ expString ^ " do " ^ "{\n" ^ stmString ^ (indent tab) ^ "}\n"
| Continue -> (indent tab) ^ "continue;"
```

```
let toStringAR (AR(h,s,f,c,r)) = (* una presentazione di AR *)
    let hString = (match h with
                  | Cmd -> "[cmd]"
                  | Exp -> "[exp]"
                  | NoneH -> "[N]"
                  | IPB -> "[IB]"
                  | WD -> "[WD]"
                  ) and
        rString = (match r with
                  | Some(aval) -> toStringAval aval
                  | _ -> "[N]"
                  ) in
    "{" ^ hString ^ "," ^ (string_of_int s) ^ "," ^ (toStringEnv f) ^
    "," ^ ":cmdNext:" ^ "," ^ rString ^ "}"
;;
```

Modifiche all'interprete

```
stmSem stm (sk,(Store(d,g)as mu)) =
  match stm with
  | WhileDo(exp,s) ->
    (match expSem exp (sk,mu) with
     |(Bool, Bval True, (ske,mue))
      ->(let h = WD in
          let ar = mkAR5 (h) (1) (emptyEnv()) ([UnL s]) (None) in
          let sk1 = push ske ar in
          let (_,(sk2,mu2)) = nextCmd (sk1,mue) in
          let skF = addCode (pop sk2) (UnL stm) in
          let sgF = (skF,mu2) in
          (Void,sgF))
     |(Bool, Bval False, sge)
      ->(Void,sge)
     | _ -> raise(TypeErrorS("E1':stmSem",stm)))
  | Continue ->
    (let h = getH sk in
     match h with
     | WD -> let skF = resetC sk [] in
              (Void,(skF,mu))
     | _ -> (let sk1 = pop sk in
              match sk1 with
              | (Stack []) -> raise(StaticErrors("Wrong Use of Continue: stmSem",stm))
              | _ -> (let skF = resetC sk1 [UnL Continue] in
                       (Void,(skF,mu))))))
```

Il programma es1 calcola il minimo numero di Fibonacci maggiore di 100.

```
let p =
  Prog ("es1",
    Block
      (SeqD (Var (Int, "x", N 0),
        SeqD (Var (Int, "y", N 1), Var (Int, "temp", N 0))),
      UnL
        (WhileDo (LT (Val "y", N 100),
          SeqS (Upd (Val "temp", Plus (Val "x", Val "y")),
            SeqS (Upd (Val "x", Val "y"), Upd (Val "y", Val "temp"))))))));;
```

```
Program es1 {
  int x = 0, y = 1;
  int temp = 0;
  while (y < 100) do {
    temp = x+y;
    x = y;
    y = temp;
  }
}
```

```
Stack:
>{es1,0,[temp/(Mint,L2);
  y/(Mint,L1);
  x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-89,L1<-144,L2<-144]
SUCCESSFUL_TERMINATION
- : unit/2 = 0
```

Verifichiamo l'evoluzione dello stack per il programma es2.

```
let p =
  Prog ("es2",
    Block (Var (Int, "x", N 0),
      UnL
        (WhileDo (LT (Val "x", N 3),
          SeqS
            (IfT (Eq (Val "x", N 1),
              SeqS (Upd (Val "x", Plus (Val "x", N 2)), Continue)),
              Upd (Val "x", Plus (Val "x", N 1))))))));;
```

```
Program es2 {
  int x = 0;
  while (x < 3) do {
    if (x == 1) {
      x = x + 2;
      continue;
    }
    x = x + 1;
  }
}
```

Verifica del codice ed esempi

```
Stack:
>{es2,0,[x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-0]

Stack:
>{[WD],1,[],:cmdNext:[N]}
{es2,0,[x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-0]

Stack:
>{[WD],1,[],:cmdNext:[N]}
{es2,0,[x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-1]

Stack:
>{es2,0,[x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-1]
```

```
Stack:
>{[WD],1,[],:cmdNext:[N]}
{es2,0,[x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-3]

Stack:
>{[WD],1,[],:cmdNext:[N]}
{es2,0,[x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-3]

Stack:
>{es2,0,[x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-3]

Stack:
>{es2,0,[x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-3]

SUCCESSFUL_TERMINATION
- : unit/2 = ()
```


Verifica del codice ed esempi

Nel programma es3 ci sono due **while-do** annidati. Il **continue** si riferisce al costrutto iterativo più interno.

```
let p =
  Prog ("es3",
    Block (SeqD (Var (Int, "x", N 10), Var (Int, "y", N 0)),
      UnL
        (WhileDo (GT (Val "x", N 7),
          SeqS
            (WhileDo (LT (Val "y", N 3),
              SeqS
                (IfT (Eq (Val "y", N 2),
                  SeqS (Upd (Val "y", Plus (Val "y", N 2)), Continue)),
                  Upd (Val "y", Plus (Val "y", N 1))))),
              Upd (Val "x", Minus (Val "x", N 1))))))));;
```

```
Program es3 {
  int x = 10;
  int y = 0;
  while (x > 7) do {
    while (y < 3) do {
      if (y == 2) {
        y = y + 2;
        continue;
      }
      y = y + 1;
    }
    x = x - 1;
  }
}
```

Verifica del codice ed esempi

Si può verificare l'effetto del **continue** analizzando l'evoluzione dello stack.

```
Stack:
>{[WD],1,[],:cmdNext:[N]}
{es3,0,[y/(Mint,L1);
x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-10,L1<-2]

Stack:
>{[WD],1,[],:cmdNext:[N]}
{[WD],1,[],:cmdNext:[N]}
{es3,0,[y/(Mint,L1);
x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-10,L1<-4]

Stack:
>{[WD],1,[],:cmdNext:[N]}
{[WD],1,[],:cmdNext:[N]}
{es3,0,[y/(Mint,L1);
x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-10,L1<-4]

Stack:
>{[WD],1,[],:cmdNext:[N]}
{es3,0,[y/(Mint,L1);
x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-10,L1<-4]
```

```
Stack:
>{[WD],1,[],:cmdNext:[N]}
{es3,0,[y/(Mint,L1);
x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-10,L1<-4]

Stack:
>{[WD],1,[],:cmdNext:[N]}
{es3,0,[y/(Mint,L1);
x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-9,L1<-4]

Stack:
>{es3,0,[y/(Mint,L1);
x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-9,L1<-4]

Stack:
>{[WD],1,[],:cmdNext:[N]}
{es3,0,[y/(Mint,L1);
x/(Mint,L0)],:cmdNext:[N]}
]
Store:
[L0<-9,L1<-4]
```



Nel programma `es4` è presente un uso scorretto del comando `continue`.

```
let p =  
  Prog ("prova1",  
    Block (Var (Int, "x", N 0),  
      UnL (BlockS (ED, SeqS (Continue, Upd (Val "x", Plus (Val "x", N 1))))))));
```

```
Program es4 {  
  int x = 0; {  
    continue;  
    x = (x + 1);  
  }  
}
```

```
# progSem p;;  
Exception: StaticErrorS ("Wrong Use of Continue: stmSem", Continue).
```

Grazie per l'attenzione!

-  Marco Bellia, 2021. *Materiale delle lezioni del corso di LPL*
-  Programming Languages: Principles and Paradigms (2010), Gabrielli M., S. Martini, Springer-Verlag, London.