

While' Tutorial

Kyle Ross

<kyle@cs.chalmers.se>

Introduction

While' is a simple imperative language based on *While* from the Nielson, Nielson, Hankin “Principles of Program Analysis” book. The following are intended to quickly introduce the reader to the constructs of the language by example. Following each example, the reduction steps that are taken by the current implementation of the *While'* interpreter are shown; these can be ignored by readers uninterested in the details. Those, however, who would like to *fully* understand the workings of the language are encouraged strongly to read the *While'* semantics, which is fairly straightforward.

Statements

Basic Statements

Statements form the heart of *While'*. The following are the basic statements inherited from *While*:

Variable assignment ($:=$)

$v := e$ binds variable v to the result of expression e

e.g., $x := 3 \Rightarrow \{x \mapsto 3\}$

$\langle x := 3, [] \rangle$

$P[| x |] [] \Rightarrow x$

$[x \rightarrow 3]$

Sequencing ($;$)

$t_1; t_2$ evaluates statement t_1 and then statement t_2

e.g., $x := 3 ; y := 4 \Rightarrow \{x \mapsto 3, y \mapsto 4\}$

$\langle x := 3 ; y := 4, [] \rangle$

$\langle x := 3, [] \rangle$

$P[| x |] [] \Rightarrow x$

$[x \rightarrow 3]$

$\langle y := 4, [x \rightarrow 3] \rangle$

$P[\mid y \mid] [x \rightarrow 3] \Rightarrow y$
 $[x \rightarrow 3, y \rightarrow 4]$

If-then-else (**if then else end**)

if e then t_1 else t_2 end evaluates expression e ; if e evaluates to T then evaluates statement t_1 ; if e evaluates to F then evaluates statement t_2

if $2 < 3$ **then** $x := 1$ **else** $x := 0$ **end** $\Rightarrow \{x \mapsto 1\}$

\langle if $(2 < 3)$ **then** $x := 1$ **else** $x := 0$ **end**, $[\] \rangle$

$E[\mid (2 < 3) \mid] [\]$

$\Rightarrow T$

$\langle x := 1, [\] \rangle$

$P[\mid x \mid] [\] \Rightarrow x$

$[x \rightarrow 1]$

if $2 > 3$ **then** $x := 1$ **else** $x := 0$ **end** $\Rightarrow \{x \mapsto 0\}$

\langle if $(2 > 3)$ **then** $x := 1$ **else** $x := 0$ **end**, $[\] \rangle$

$E[\mid (2 > 3) \mid] [\]$

$\Rightarrow F$

$\langle x := 0, [\] \rangle$

$P[\mid x \mid] [\] \Rightarrow x$

$[x \rightarrow 0]$

Looping

Unlike *While*, *While'* does *not* have general looping constructs. The sub-recursive looping constructs provided are:

Integer looping (**for**)

for v from n do t end evaluates t where $v := n$ then where $v := n - 1$ then ... then where $v := 1$

e.g., $s := 0$; **for** i **from** 3 **do** $s := s + i$ **end** $\Rightarrow \{s \mapsto 6\}$

$\langle s := 0$; **for** i **from** 3 **do** $s := (s + i)$ **end**, $[s \rightarrow 6] \rangle$

$\langle s := 0, [s \rightarrow 6] \rangle$

$P[\mid s \mid] [s \rightarrow 6]$

$\Rightarrow s$

$[s \rightarrow 0]$

\langle **for** i **from** 3 **do** $s := (s + i)$ **end**, $[s \rightarrow 0] \rangle$

$\langle s := (s + i), [s \rightarrow 0, i \rightarrow 3] \rangle$

$E[\mid (s + i) \mid] [s \rightarrow 0, i \rightarrow 3]$

$E[\mid s \mid] [s \rightarrow 0, i \rightarrow 3] \Rightarrow 0$

$E[\mid i \mid] [s \rightarrow 0, i \rightarrow 3] \Rightarrow 3$

$\Rightarrow 3$

$P[\mid s \mid] [s \rightarrow 0, i \rightarrow 3]$

```

=> s
[s->3, i->3]
<for i from 2 do s := (s + i) end, [s->3, i->3]>
<s := (s + i), [s->3, i->2]>
  E[| (s + i) |] [s->3, i->2]
    E[| s |] [s->3, i->2] => 3
    E[| i |] [s->3, i->2] => 2
  => 5
  P[| s |] [s->3, i->2]
  => s
[s->5, i->2]
<for i from 1 do s := (s + i) end, [s->5, i->2]>
<s := (s + i), [s->5, i->1]>
  E[| (s + i) |] [s->5, i->1]
    E[| s |] [s->5, i->1] => 5
    E[| i |] [s->5, i->1] => 1
  => 6
  P[| s |] [s->5, i->1]
  => s
[s->6, i->1]
<for i from 0 do s := (s + i) end, [s->6, i->1]>
[s->6, i->1]
[s->6, i->2]
[s->6, i->3]
[s->6]
e.g., x := 0 ; for i from 3 do i := 5 ; x := i end => {x ↦ 5}
<x := 0 ; for i from 3 do i := 5 ; x := i end, []>
<x := 0, []>
  P[| x |] [] => x
[x->0]
<for i from 3 do i := 5 ; x := i end, [x->0]>
<i := 5 ; x := i, [x->0, i->3]>
<i := 5, [x->0, i->3]>
  P[| i |] [x->0, i->3] => i
[x->0, i->5]
<x := i, [x->0, i->5]>
  E[| i |] [x->0, i->5] => 5
  P[| x |] [x->0, i->5] => x
[x->5, i->5]
<for i from 2 do i := 5 ; x := i end, [x->5, i->5]>
<i := 5 ; x := i, [x->5, i->2]>
<i := 5, [x->5, i->2]>
  P[| i |] [x->5, i->2] => i
[x->5, i->5]
<x := i, [x->5, i->5]>

```

```

E[| i |] [x->5, i->5] => 5
P[| x |] [x->5, i->5] => x
[x->5, i->5]
<for i from 1 do i := 5 ; x := i end, [x->5, i->5]>
<i := 5 ; x := i, [x->5, i->1]>
<i := 5, [x->5, i->1]>
P[| i |] [x->5, i->1] => i
[x->5, i->5]
<x := i, [x->5, i->5]>
E[| i |] [x->5, i->5] => 5
P[| x |] [x->5, i->5] => x
[x->5, i->5]
<for i from 0 do i := 5 ; x := i end, [x->5, i->5]>
[x->5, i->5]
[x->5]

```

Structural recursion over lists

foreach v in l do t end evaluates t where $v := l^{th} 1$ then where $v := l^{th} 2$ then ... then where $v := l^{th} (length\ l)$

e.g., $s := 0$; **foreach** i in $[2, 3, 5]$ **do** $s := s + i$ **end** $\Rightarrow \{s \mapsto 10\}$

```

<s := 0 ; foreach i in [2, 3, 5] do s := (s + i) end, []>
<s := 0, []>
P[| s |] [] => s
[s->0]
<foreach i in [2, 3, 5] do s := (s + i) end, [s->0]>
<s := (s + i), [s->0, i->2]>
E[| (s + i) |] [s->0, i->2]
E[| s |] [s->0, i->2] => 0
E[| i |] [s->0, i->2] => 2
=> 2
P[| s |] [s->0, i->2] => s
[s->2, i->2]
<foreach i in [3, 5] do s := (s + i) end, [s->2, i->2]>
<s := (s + i), [s->2, i->3]>
E[| (s + i) |] [s->2, i->3]
E[| s |] [s->2, i->3] => 2
E[| i |] [s->2, i->3] => 3
=> 5
P[| s |] [s->2, i->3] => s
[s->5, i->3]
<foreach i in [5] do s := (s + i) end, [s->5, i->3]>
<s := (s + i), [s->5, i->5]>
E[| (s + i) |] [s->5, i->5]
E[| s |] [s->5, i->5] => 5

```

```

E[| i |] [s->5, i->5] => 5
=> 10
P[| s |] [s->5, i->5] => s
[s->10, i->5]
<foreach i in [] do s := (s + i) end, [s->10, i->5]>
[s->10, i->5]
[s->10, i->3]
[s->10, i->2]
[s->10]

```

References

While' extends *While* by adding named references (aliases):

Reference creation

$v := \text{ref } v'$ binds v to a reference to v'
e.g., $r := \#x \Rightarrow \{r \mapsto x\# \}$
 $\langle r := \#x, [] \rangle$
 $P[| r |] [] \Rightarrow r$
 $[r \rightarrow \#x]$

Expression dereference

$!r$ returns the value bound to the variable to which r refers
e.g., $x := 3 ; r := \#x ; y := !r \Rightarrow \{r \mapsto x\#, x \mapsto 3, y \mapsto 3\}$
 $\langle x := 3 ; r := \#x ; y := (! r), [] \rangle$
 $\langle x := 3, [] \rangle$
 $P[| x |] [] \Rightarrow x$
 $[x \rightarrow 3]$
 $\langle r := \#x ; y := (! r), [x \rightarrow 3] \rangle$
 $\langle r := \#x, [x \rightarrow 3] \rangle$
 $P[| r |] [x \rightarrow 3] \Rightarrow r$
 $[x \rightarrow 3, r \rightarrow \#x]$
 $\langle y := (! r), [x \rightarrow 3, r \rightarrow \#x] \rangle$
 $E[| (! r) |] [x \rightarrow 3, r \rightarrow \#x]$
 $E[| r |] [x \rightarrow 3, r \rightarrow \#x] \Rightarrow \#x$
 $\Rightarrow 3$
 $P[| y |] [x \rightarrow 3, r \rightarrow \#x] \Rightarrow y$
 $[x \rightarrow 3, r \rightarrow \#x, y \rightarrow 3]$

Pattern dereference

$r!$ returns the variable to which r refers
e.g., $r := \#x ; r! := 3 \Rightarrow \{r \mapsto x\#, x \mapsto 3\}$

```

<r := #x ; (r !) := 3, []>
<r := #x, []>
P[! r |] [] => r
[r->#x]
<(r !) := 3, [r->#x]>
P[! (r) |] [r->#x]
P[! r |] [r->#x] => r
E[! r |] [r->#x] => #x
=> x
[r->#x, x->3]

```

Structures

While' also extends *While* by adding simple structures:

Structure creation

$v := \text{newstruct}$ binds variable v to an empty, extensible structure

e.g., $x := \text{newstruct} \Rightarrow \{x \mapsto \{\}\}$

```

<x := newstruct, []>
E[! newstruct |] []
R[! newstruct |]
{}
=> {}
P[! x |] [] => x
[x->{}]

```

Structure update

$v.f := e$ binds the f field of the structure stored in variable v to the result of expression e

e.g., $x := \text{newstruct} ; x.a := 3 \Rightarrow \{x \mapsto \{a \mapsto 3\}\}$

```

<x := newstruct ; x.a := 3, []>
<x := newstruct, []>
E[! newstruct |] []
R[! newstruct |]
{}
=> {}
P[! x |] [] => x
[x->{}]
<x.a := 3, [x->{}]>
P[! x.a |] [x->{}]
P[! x |] [x->{}] => x
=> x.a
P[! x |] [x->{}] => x
E[! x |] [x->{}] => {}

```

```

R[[] {}[a |-> 3 ]]
{a->3}
<x := {a->3}, [x->{}]>
P[[] x []] [x->{}] => x
[x->{a->3}]

```

Expression structure dereference

!r.f returns the value bound to the variable to which the f field of r refers

e.g., $y := 3 ; x := \text{newstruct} ; x.a := \#y ; z := !(x.a) \Rightarrow \{x \mapsto \{a \mapsto y\}, y \mapsto 3, z \mapsto 3\}$

```

<y := 3 ; x := newstruct ; x.a := #y ; z := (! x.a), []>

```

```

<y := 3, []>
P[[] y []] [] => y
[y->3]

```

```

<x := newstruct ; x.a := #y ; z := (! x.a), [y->3]>

```

```

<x := newstruct, [y->3]>

```

```

E[[] newstruct []] [y->3]

```

```

R[[] newstruct []]

```

```

{}

```

```

=> {}

```

```

P[[] x []] [y->3] => x

```

```

[y->3, x->{}]

```

```

<x.a := #y ; z := (! x.a), [y->3, x->{}]>

```

```

<x.a := #y, [y->3, x->{}]>

```

```

P[[] x.a []] [y->3, x->{}]

```

```

P[[] x []] [y->3, x->{}] => x

```

```

=> x.a

```

```

P[[] x []] [y->3, x->{}] => x

```

```

E[[] x []] [y->3, x->{}] => {}

```

```

R[[] {}[a |-> #y ]]

```

```

{a->#y}

```

```

<x := {a->#y}, [y->3, x->{}]>

```

```

P[[] x []] [y->3, x->{}] => x

```

```

[y->3, x->{a->#y}]

```

```

[y->3, x->{a->#y}]

```

```

<z := (! x.a), [y->3, x->{a->#y}]>

```

```

E[[] (! x.a) []] [y->3, x->{a->#y}]

```

```

E[[] x.a []] [y->3, x->{a->#y}]

```

```

E[[] x []] [y->3, x->{a->#y}] => {a->#y}

```

```

=> #y

```

```

=> 3

```

```

P[[] z []] [y->3, x->{a->#y}] => z

```

```

[y->3, x->{a->#y}, z->3]

```

$(!r).f$ returns the value bound to the f field in the structure bound to the variable to which r refers

e.g., $x := \text{newstruct} ; x.a := 3 ; y := \#x ; z := !y.a \Rightarrow \{x \mapsto \{a \mapsto 3\}, y \mapsto x\#, z \mapsto 3\}$

```

<x := newstruct ; x.a := 3 ; y := #x ; z := (! y).a, []>
<x := newstruct, []>
  E[| newstruct |] []
  R[| newstruct |]
  {}
  => {}
  P[| x |] [] => x
[x->{}]
<x.a := 3 ; y := #x ; z := (! y).a, [x->{}]>
<x.a := 3, [x->{}]>
  P[| x.a |] [x->{}]
  P[| x |] [x->{}] => x
  => x.a
  P[| x |] [x->{}] => x
  E[| x |] [x->{}] => {}
  R[| {}[a |-> 3 |]
  {a->3}
  <x := {a->3}, [x->{}]>
  P[| x |] [x->{}] => x
  [x->{a->3}]
<y := #x ; z := (! y).a, [x->{a->3}]>
<y := #x, [x->{a->3}]>
  P[| y |] [x->{a->3}] => y
  [x->{a->3}, y->#x]
<z := (! y).a, [x->{a->3}, y->#x]>
  E[| (! y).a |] [x->{a->3}, y->#x]
  E[| (! y) |] [x->{a->3}, y->#x]
  E[| y |] [x->{a->3}, y->#x] => #x
  => {a->3}
  => 3
  P[| z |] [x->{a->3}, y->#x] => z
[x->{a->3}, y->#x, z->3]

```

Pattern structure dereference

$r!.f := e$ binds the f field of the structure to which r refers to the result of expression e

e.g., $x := \text{newstruct} ; y := \#x ; y!.a := 3 \Rightarrow \{x \mapsto \{a \mapsto 3\}, y \mapsto x\# \}$

```

<x := newstruct ; y := #x ; (y !).a := 3, []>
<x := newstruct, []>

```

```

E[| newstruct |] []
R[| newstruct |]
{}
=> {}
P[| x |] [] => x
[x->{}]
<y := #x ; (y !).a := 3, [x->{}]>
<y := #x, [x->{}]>
P[| y |] [x->{}] => y
[x->{}, y->#x]
<(y !).a := 3, [x->{}, y->#x]>
P[| (! y).a |] [x->{}, y->#x]
P[| (! y) |] [x->{}, y->#x]
P[| y |] [x->{}, y->#x] => y
E[| y |] [x->{}, y->#x] => #x
=> x
=> x.a
P[| x |] [x->{}, y->#x] => x
E[| x |] [x->{}, y->#x] => {}
R[| {}[a |-> 3 |]
{a->3}
<x := {a->3}, [x->{}, y->#x]>
P[| x |] [x->{}, y->#x] => x
[x->{a->3}, y->#x]

```

e.g., $x := \text{newstruct} ; x.a := \#y ; x.a! := 3 \Rightarrow \{x \mapsto \{a \mapsto y\}, y \mapsto 3\}$

```

<x := newstruct ; x.a := #y ; (x.a !).a := 3, []>
<x := newstruct, []>
E[| newstruct |] []
R[| newstruct |]
{}
=> {}
P[| x |] [] => x
[x->{}]
<x.a := #y ; (x.a !).a := 3, [x->{}]>
<x.a := #y, [x->{}]>
P[| x.a |] [x->{}]
P[| x |] [x->{}] => x
=> x.a
P[| x |] [x->{}] => x
E[| x |] [x->{}] => {}
R[| {}[a |-> #y |]
{a->#y}
<x := {a->#y}, [x->{}]>
P[| x |] [x->{}] => x

```

```

[x->{a->#y}]
[x->{a->#y}]
<(x.a !) := 3, [x->{a->#y}]>
P[| (! x.a) |] [x->{a->#y}]
P[| x.a |] [x->{a->#y}]
P[| x |] [x->{a->#y}] => x
=> x.a
E[| x.a |] [x->{a->#y}]
E[| x |] [x->{a->#y}] => {a->#y}
=> #y
=> y
[x->{a->#y}, y->3]

```

Let

While' additionally extends *While* with a simple local assignment:

Local assignment

let $v := e$ in s end binds variable v the result of evaluating expression e before evaluating statement s ; then the previous binding for v is restored

e.g., $x := 2$; let $x := 5$ in $y := x + 3$ end $\Rightarrow \{x \mapsto 2, y \mapsto 8\}$

< $x := 2$; let $x = 5$ in $y := (x + 3)$ end, []>

< $x := 2$, []>

P[| x |] [] => x

[$x \rightarrow 2$]

<let $x = 5$ in $y := (x + 3)$ end, [$x \rightarrow 2$]>

< $y := (x + 3)$, [$x \rightarrow 5$]>

E[| $(x + 3)$ |] [$x \rightarrow 5$]

E[| x |] [$x \rightarrow 5$] => 5

=> 8

P[| y |] [$x \rightarrow 5$] => y

[$x \rightarrow 5$, $y \rightarrow 8$]

[$x \rightarrow 2$, $y \rightarrow 8$]

Expressions

Arithmetic Operators

Addition

$x + y$ returns the sum of integers x and y

e.g., $x := 2 + 3 \Rightarrow \{x \mapsto 5\}$

```

<x := (2 + 3), []>
E[| (2 + 3) |] []
=> 5
P[| x |] [] => x
[x->5]

```

Subtraction

$x - y$ returns the difference of integers x and y

e.g., $x := 3 - 2 \Rightarrow \{x \mapsto 1\}$

```

<x := (3 - 2), []>
E[| (3 - 2) |] []
=> 1
P[| x |] [] => x
[x->1]

```

Multiplication

$x * y$ returns the product of integers x and y

e.g., $x := 2 * 3 \Rightarrow \{x \mapsto 6\}$

```

<x := (2 * 3), []>
E[| (2 * 3) |] []
=> 6
P[| x |] [] => x
[x->6]

```

Division

x/y returns the (integer) ratio of integers x and y

e.g., $x := 2 / 3 \Rightarrow \{x \mapsto 0\}$

```

<x := (2 / 3), []>
E[| (2 / 3) |] []
=> 0
P[| x |] [] => x
[x->0]

```

Remainder

$x \bmod y$ returns the remainder when integer x is divided by integer y

e.g., $x := 3 \bmod 2 \Rightarrow \{x \mapsto 1\}$

```

<x := (3 mod 2), []>
E[| (3 mod 2) |] []
=> 1
P[| x |] [] => x
[x->1]

```

Unary negation

neg x returns $0 - x$
e.g., $x := \mathbf{neg\ 2} \Rightarrow \{x \mapsto -2\}$
 $\langle x := (\mathbf{neg\ 2}), [] \rangle$
 $E[[] (\mathbf{neg\ 2}) [] []]$
 $\Rightarrow -2$
 $P[[] x [] []] \Rightarrow x$
 $[x \rightarrow -2]$

Logical Operators

Conjunction

x and y returns the conjunction of Booleans x and y
e.g., $x := \mathbf{T\ and\ F} \Rightarrow \{x \mapsto F\}$
 $\langle x := (\mathbf{T\ and\ F}), [] \rangle$
 $E[[] (\mathbf{T\ and\ F}) [] []]$
 $\Rightarrow F$
 $P[[] x [] []] \Rightarrow x$
 $[x \rightarrow F]$

Disjunction

x or y returns the disjunction of Booleans x and y
e.g., $x := \mathbf{T\ or\ F} \Rightarrow \{x \mapsto T\}$
 $\langle x := (\mathbf{T\ or\ F}), [] \rangle$
 $E[[] (\mathbf{T\ or\ F}) [] []]$
 $\Rightarrow T$
 $P[[] x [] []] \Rightarrow x$
 $[x \rightarrow T]$

Negation

$\neg x$ returns the negation of Boolean x
e.g., $x := \mathbf{\neg T} \Rightarrow \{x \mapsto F\}$
 $\langle x := (\mathbf{\sim T}), [] \rangle$
 $E[[] (\mathbf{\sim T}) [] []]$
 $\Rightarrow F$
 $P[[] x [] []] \Rightarrow x$
 $[x \rightarrow F]$

List Operators

Element addition

$x :: y$ returns the addition of value x to list y

e.g., $x := 3 :: [4, 5] \Rightarrow \{x \mapsto [3, 4, 5]\}$

$\langle x := (3 :: [4, 5]), [] \rangle$

$E[(3 :: [4, 5]) \mid] []$

$\Rightarrow [3, 4, 5]$

$P[x \mid] [] \Rightarrow x$

$[x \rightarrow [3, 4, 5]]$

Concatenation

$x ++ y$ returns the concatenation of lists x and y

e.g., $x := [1, 2] ++ [3, 4] \Rightarrow \{x \mapsto [1, 2, 3, 4]\}$

$\langle x := ([1, 2] ++ [3, 4]), [] \rangle$

$E[([1, 2] ++ [3, 4]) \mid] []$

$\Rightarrow [1, 2, 3, 4]$

$P[x \mid] [] \Rightarrow x$

$[x \rightarrow [1, 2, 3, 4]]$

Element access

$x \text{ nth } y$ returns the y^{th} element of list x for an integer index y

e.g., $x := [5, 4, 3] \text{ nth } 2 \Rightarrow \{x \mapsto 4\}$

$\langle x := ([5, 4, 3] \text{ nth } 2), [] \rangle$

$E[([5, 4, 3] \text{ nth } 2) \mid] []$

$\Rightarrow 4$

$P[x \mid] [] \Rightarrow x$

$[x \rightarrow 4]$

Head access

$\text{hd } x$ returns the first element of a list x

e.g., $x := \text{hd } [5, 4, 3] \Rightarrow \{x \mapsto 5\}$

$\langle x := (\text{hd } [5, 4, 3]), [] \rangle$

$E[(\text{hd } [5, 4, 3]) \mid] []$

$\Rightarrow 5$

$P[x \mid] [] \Rightarrow x$

$[x \rightarrow 5]$

Tail access

$\text{tl } x$ returns a list x without its first element

e.g., $x := \text{tl } [5, 4, 3] \Rightarrow \{x \mapsto [4, 3]\}$

```

<x := (tl [5, 4, 3]), []>
E[| (tl [5, 4, 3]) |] []
=> [4, 3]
P[| x |] [] => x
[x->[4, 3]]

```

Flatten

flatten x returns a fully-flattened version of x

e.g., $x := \mathbf{flatten}$ [1, [2], [[3]], [[[4]]], [[[[5, [[6]]]]]] \Rightarrow
 $\{x \mapsto [1, 2, 3, 4, 5, 6]\}$

```

<x := (flatten [1, [2], [[3]], [[[4]]], [[[[5, [[6]]]]]]), []>
E[| (flatten [1, [2], [[3]], [[[4]]], [[[[5, [[6]]]]]]) |] []
=> [1, 2, 3, 4, 5, 6]
P[| x |] [] => x
[x->[1, 2, 3, 4, 5, 6]]

```

Reverse

rev x returns a list with the same elements as x but in reverse order

e.g., $x := \mathbf{rev}$ [5, 4, 3] \Rightarrow $\{x \mapsto [3, 4, 5]\}$

```

<x := (rev [5, 4, 3]), []>
E[| (rev [5, 4, 3]) |] []
=> [3, 4, 5]
P[| x |] [] => x
[x->[3, 4, 5]]

```

n.b., $y := \mathbf{rev}$ x could be written as: $y := []$; **foreach** i **in** x **do** $y := i::y$ **end**

Length

length x returns the length of list x

e.g., $x := \mathbf{length}$ [4, 5, 6] \Rightarrow $\{x \mapsto 3\}$

```

<x := (length [4, 5, 6]), []>
E[| (length [4, 5, 6]) |] []
=> 3
P[| x |] [] => x
[x->3]

```

n.b., $y := \mathbf{length}$ x could be written as: $y := 0$; **foreach** i **in** x **do** $y := y + 1$ **end**

Comparison Operators

Equality comparison

$x = y$ returns whether values x and y are equal (x and y need not be the same type and can be structures or references)

e.g., $x := 1 = 2 \Rightarrow \{x \mapsto F\}$

$\langle x := (1 = 2), [] \rangle E[\text{---} (1 = 2) \text{---}] [] =_i F \quad P[\text{---} x \text{---}] [] =_i x$
 $[x \mapsto F]$

e.g., $x := \mathbf{newstruct}; y := \mathbf{newstruct}; x.a := 3; x.b := 4; y.b := 4; y.a := 3; z := x = y \Rightarrow \{x \mapsto \{a \mapsto 3, b \mapsto 4\}, y \mapsto \{a \mapsto 3, b \mapsto 4\}, z \mapsto T\}$

$\langle z := (x = y), [x \mapsto \{b \mapsto 4, a \mapsto 3\}, y \mapsto \{a \mapsto 3, b \mapsto 4\}] \rangle$
 $E[| (x = y) |] [x \mapsto \{b \mapsto 4, a \mapsto 3\}, y \mapsto \{a \mapsto 3, b \mapsto 4\}]$
 $E[| x |] [x \mapsto \{b \mapsto 4, a \mapsto 3\}, y \mapsto \{a \mapsto 3, b \mapsto 4\}] \Rightarrow \{b \mapsto 4, a \mapsto 3\}$
 $E[| y |] [x \mapsto \{b \mapsto 4, a \mapsto 3\}, y \mapsto \{a \mapsto 3, b \mapsto 4\}] \Rightarrow \{a \mapsto 3, b \mapsto 4\}$
 $\Rightarrow T$
 $P[| z |] [x \mapsto \{b \mapsto 4, a \mapsto 3\}, y \mapsto \{a \mapsto 3, b \mapsto 4\}] \Rightarrow z$
 $[x \mapsto \{b \mapsto 4, a \mapsto 3\}, y \mapsto \{a \mapsto 3, b \mapsto 4\}, z \mapsto T]$

(this shows the evaluation of only the last statement (viz., the assignment to z))

Less-than comparison

$x < y$ returns whether integer x is less than integer y

e.g., $x := 1 < 2 \Rightarrow \{x \mapsto T\}$

$\langle x := (1 < 2), [] \rangle$
 $E[| (1 < 2) |] []$
 $\Rightarrow T$
 $P[| x |] [] \Rightarrow x$
 $[x \mapsto T]$

Greater-than comparison

$x > y$ returns whether integer x is greater than integer y

e.g., $x := 1 > 2 \Rightarrow \{x \mapsto F\}$

$\langle x := (1 > 2), [] \rangle$
 $E[| (1 > 2) |] []$
 $\Rightarrow F$
 $P[| x |] [] \Rightarrow x$
 $[x \mapsto F]$

Examples

Factorial

Of course, when learning a new programming language the first thing one would like to do is to write factorial. Although *While'* is sub-recursive, this is possible via iteration over integers:

```

x := 5
fac_x := 1
for i from x do fac_x := fac_x * i end

```

Fibonacci sequence

We can generate (a finite prefix of) the Fibonacci sequence by initialising a list `fibs` with the first two values:

```
fibs := [1, 1]
```

Then, we append a new element at the end of `fibs`. We calculate this element by reversing `fibs` and adding the first two elements from the reversed list:

```

l := 10
for i from l do t := rev fibs ; fibs := fibs ++ [hd t + (t
  nth 2)] end

```

A more efficient loop body would be:

```

l := 10
for i from l do ind := l + 1 - i ; fibs := fibs ++ [(fibs
  nth ind) + (fibs nth (ind + 1))] end

```

The first version, however, has the advantage that we can run the same loop on the result to obtain the next `l` elements in the sequence.

Circularly-linked list

Another fun example is the circularly-linked list. First, we create 3 “cons” cells:

```

a1 := newstruct
a2 := newstruct
a3 := newstruct

```

Next, we set the “car” (value) of each cell:

```

a1.a := 1
a2.a := 2
a3.a := 3

```

Then, we link them in a circle by setting the “cdr” (next) of each cell, linking the last cell to the first:

```

a1.b := #a2
a2.b := #a3
a3.b := #a1

```

We set the current pointer to the first cell and initialise the sum to 0:

```

i := #a1
s := 0

```

Then we iterate over the cells for 100 times (this was a completely arbitrary choice), adding the “car” value of the current cell to the sum and following its “cdr-chain”:

```
for j from 100 do s := s + (!i).a ; i := (!i).b end
```