Whiley Cheat Sheet

By David J. Pearce, 2014. See `http://whiley.org`

Values

Values are the fundamental units of execution in Whiley and have value semantics, rather than reference semantics (as in many object-oriented languages).

`null`	*Null value*
`true` `false`	*Boolean values*
`123` `-99` `0xFF`	*Integer values*
`1.23` `-0.02`	*Real values*
`"Hello"` `"new\n line"`	*String values*
`(1,2,3)` `(true,null)`	*Tuple values*
`[1,2,3]` `[1,"xyz"]`	*Array values*
`{name: "dave"}`, `{x: 1, y: 0}`	*Record values*

Types

The Whiley programming language is *statically typed*, meaning that every expression has a type determined at compile time. Furthermore, evaluating an expression is guaranteed to yield a value of its type.

`null` `bool` `int` `real`	*Primitive types*
`int|null` `bool|int|real`	*Union types*
`(int,int)` `(int,null,bool)`	*Tuple types*
`int[]` `bool[][]` `(int|null)[]`	*Array types*
`{bool f}` `{int len, int[] is}`	*Record types*

Expressions

The majority of work performed by a Whiley program is through the execution of *expressions*. Every expression produces a value and may have additional side effects.

`x + 1` `2 * y` `z - 1` `(x + y)/2`	*Arithmetic*
`x < y` `0 >= z` `x == y` `x != y`	*Comparisons*
`!x` `x&&y` `x||y` `x==>y` `x<==>y`	*Boolean*
`|ls|` `ls[0]` `[true; n]`	*Arrays*
`[1,x+y]` `x in xs`	
`{x: 1+y}` `xr.f` `xr.f.g`	*Records*
`all { i in 0..|xs| | xs[i]>= 0}`	*Quantifiers*
`no { i in 0..|xs| | xs[i]<0 }`	
`some { i in 0..|xs| | xs[i]>=0 }`	
`x is null` `x is int`	*Type Tests*

Statements

The execution of a Whiley program is controlled by *statements*, which cause effects on the environment. Statements in Whiley do not produce values. Compound statements may contain other statements.

Variables are declared and initialised through *variable declarations*. Variables must be declared before being used.

```
int x
```
```
int x = 1
```
```
int x, int y
```

Variables, fields and map or list elements can be *assigned*. Variables must be defined before being used.

```
x = x + y
```
```
x[0] = 1
```
```
r.f = 3
```
```
x,y = t
```

Conditional statements control the flow of execution based on the result of a boolean expression.

```
if x < 0:
   ...
...
```
```
if x < 0:
   ...
else:
   ...
```
```
if x < 0:
   ...
else if x > 0:
   ...
```

Looping statements control the flow of execution by repeating some sequence of statements zero or more times.

```
while x<0:
   ...
...
```
```
do:
   ...
while x<0
...
```

Switch statements control execution flow by matching the result of an expression.

```
switch x:
   case 1:
       x = x + 1
   case 1,2:
       x = 0
...
```
```
switch x:
   case 1:
       x = x + 1
   default:
       x = 0
...
```

Return statements terminate the execution of a function or method and may return the result of an expression.

```
return
```
```
return x + 3
```
```
return x,y
```

Assertion and *assumption* statements enable the programmer to express knowledge at a given point.

```
assert x > 0
```
```
assume x > 0 ==> y < 3
```

Break statements terminate loops early; *debug* statements enable output from functions; *skip* statements are a no-op.

```
break
```
```
debug "got here"
```
```
skip
```

Declarations

A *declaration* declares a named entity within a source file and may refer to named entities in this or other source files and (in some cases) may *recursively* refer to itself.

Constant declarations define constants with known values at compile-time (they cannot be recursively defined).

```
constant PI is 3.1459265359
constant TWO_PI is 2*3.1459265359
```

Type declarations define named types composed from other types (they may be recursively defined).

```
type Point is { int x, int y }
```

```
type Link is { LinkedList next, int data }
type LinkedList is null | Link
```

Function declarations define functions which are *pure* and may not have side-effects. They are guaranteed to return the same result given the same arguments, and are permitted within specifications.

```
function find(int[] xs, int x) -> int:
   ...
```

Method declarations define methods which are *impure* and may have side-effects. They cannot be used within specifications.

```
method main(System.Console console):
   console.out.println("Hello␣World")
```

Specifications

A *precondition* is a condition over the parameters of a function that must hold when the function is called. A *postcondition* is a condition over the return values of a function which is required to be true after the function is called.

```
function decrement(int x) -> (int y)
// Parameter x must be greater than zero
requires x > 0
// Return must be greater or equal to zero
ensures y >= 0
// Return must be less than input
ensures y > x:
    //
    return x - 1
```

A *data-type invariant* is a constraint on the values of a declared type which must be true for any instance of it.

```
type nat is (int n) where n >= 0
type pos is (int p) where p > 0
```

A *loop invariant* is a property which holds before and after each iteration of the loop, such that: **(1)** the loop invariant must hold on entry to the loop; **(2)** assuming the loop invariant holds at the start of the loop body (along with the condition), it must hold at the end; **(3)** the loop invariant (along with the negated condition) can be assumed to hold immediately after the loop.

```
...
int i = 0
while i < x where i >= 0:
    i = i + 1
...
```

Examples

The following function computes the maximum value of two integer parameters.

```
function max(int x, int y) -> (int z)
// must return either x or y
ensures x == z || y == z
// return must be as large as x and y
ensures x <= z && y <= z:
    // implementation
    if x > y:
        return x
    else:
        return y
```

The following function uses a **`break`** to exit a **`while`** loop when the first element matching parameter `x` is found.

```
// Find index of matching element, or return -1
function indexOf(int[] xs, int x) -> int:
  int i = 0
  //
  while i < |xs| where i >= 0:
     if xs[i] == x:
        return i
     i = i + 1
  return -1
```

The following function computes the length of a linked list.

```
// A linked list is either the empty list or a link
type LinkedList is null | Link
// A single link in a linked list
type Link is {int data, LinkedList next}

// Return length of linked list (i.e. number of links it contains)
function length(LinkedList l) -> int:
  if l is null:
    // l now has type null
    return 0
  else:
    // l now has type {int data, LinkedList next}
    return 1 + length(l.next)
```

The following function reverses the values in a list of integers.

```
function reverse(int[] xs) -> (int[] ys)
// size of lists are the same
ensures |xs| == |ys|:
   int i = 0
   int[] zs = xs
   //
   while i<|xs| where i>=0 && |xs|==|zs|:
       int j = |xs| - (i+1)
       xs[i] = zs[j]
       i = i + 1
   return xs
```