The Whiley Language Specification

Updated for version 0.3.33

David J. Pearce, 2014

Contents

1 Introduction **7**

1.1 Background 7

1.2 Goals 8

1.3 History 8

2 Lexical Structure **9**

2.1 Line Terminators 9

2.2 Indentation 9

2.3 Comments 10

2.4 Identifiers 10

2.5 Keywords 10

2.6 Literals 11

2.6.1 Null Literal 11

2.6.2 Boolean Literals 12

2.6.3 Byte Literals 12

2.6.4 Integer Literals 12

2.6.5 Real Literals 12

2.6.6 Character Literals 13

2.6.7 String Literals 13

3 Source Files **15**

3.1 Compilation Units 15

3.2 Packages 15

3.3 Names 16

3.4 Imports 16

3.5 Named Declarations 17

3.5.1 Access Control 17

3.5.2 Type Declarations 17

3.5.3 Constant Declarations 18

3.5.4 Function Declarations 18

3.5.5 Method Declarations 19

4 Types & Values **21**
4.1 Overview . . . 21
4.2 Type Descriptors . . . 22
4.3 Type Patterns . . . 22
4.4 Primitive Types . . . 23
4.4.1 Null . . . 23
4.4.2 Booleans . . . 24
4.4.3 Bytes . . . 24
4.4.4 Integers . . . 25
4.4.5 Rationals . . . 26
4.4.6 Any . . . 26
4.4.7 Void . . . 27
4.5 Tuples . . . 28
4.6 Records . . . 28
4.7 References . . . 29
4.8 Collections . . . 30
4.8.1 Sets . . . 30
4.8.2 Maps . . . 31
4.8.3 Lists . . . 31
4.9 Functions and Methods . . . 32
4.10 Unions . . . 33
4.11 Intersections . . . 33
4.12 Negations . . . 34
4.13 Recursive Types . . . 35
4.14 Effective Types . . . 35
4.14.1 Effective Tuples . . . 35
4.14.2 Effective Records . . . 35
4.14.3 Effective Collections . . . 36
4.15 Semantics . . . 36
4.15.1 Equivalences . . . 36
4.15.2 Subtyping . . . 36

5 Statements **37**
5.1 Blocks . . . 37
5.2 Assert Statement . . . 38
5.3 Assignment Statement . . . 38
5.4 Assume Statement . . . 39
5.5 Break Statement . . . 39
5.6 Continue Statement . . . 40
5.7 Debug Statement . . . 41
5.8 Do/While Statement . . . 41
5.9 For Statement . . . 42
5.10 Fail Statement . . . 43

5.11 If Statement 44
5.12 While Statement 44
5.13 Return Statement 45
5.14 Skip Statement 46
5.15 Switch Statement 46
5.16 Variable Declaration Statement 47

6 Expressions **49**
6.1 Evaluation Order 49
6.2 Multi Expressions 49
6.3 Unit Expressions 49
6.4 Logical Expressions 50
6.5 Quantifier Expressions 51
6.6 Binary Expressions 51
6.6.1 Logical Expressions 51
6.6.2 Relational Expressions 52
6.6.3 Bitwise Expressions 53
6.6.4 Set Expressions 54
6.6.5 Append Expressions 54
6.6.6 Range Expressions 55
6.6.7 Shift Expressions 55
6.6.8 Additive Expressions 56
6.6.9 Multiplicative Expressions 56
6.7 Access Expressions 56
6.8 Unary Expressions 57
6.8.1 Negation Expression 57
6.8.2 LengthOf Expression 58
6.8.3 Bracketed Expression 58
6.8.4 Cast Expression 58
6.8.5 Logical Not Expression 58
6.8.6 Bitwise Complement Expression 59
6.8.7 New Expression 59
6.8.8 Dereference Expression 59
6.9 Constructor Expressions 60
6.9.1 Set Constructors 60
6.9.2 Map Constructors 60
6.9.3 List Constructors 60
6.9.4 Record Constructors 60
6.10 Term Expressions 61
6.11 Lambda Expression 61

7 Flow Typing **63**

8 Definite Assignment **65**

9 Verification **67**

10 Error Messages **69**
10.1 Parse Errors . . . 69
10.2 Declarations . . . 69
10.2.1 “Cyclic Constant Declaration” (301) . . . 69
10.3 Expressions . . . 70
10.3.1 “Invalid File Access” (000) . . . 70
10.3.2 “Invalid Package Access” (000) . . . 70
10.3.3 “Resolution Error” (000) . . . 70
10.3.4 “Variable Possibly Uninitialised” (000) . . . 70
10.3.5 “Ambiguous Coercion” (000) . . . 70
10.4 Control Flow (000) . . . 70
10.4.1 “Invalid LVal” (000) . . . 70
10.4.2 “Invalid Tuple LVal” (000) . . . 70
10.4.3 “Break Outside of Loop” (000) . . . 70
10.4.4 “Unknown Variable” (000) . . . 70
10.4.5 “Unknown Function or Method” (000) . . . 70
10.4.6 “Variable Already Defined” (000) . . . 70
10.4.7 “Duplicate Default Label” (000) . . . 71
10.4.8 “Duplicate Case Label” (000) . . . 71
10.4.9 “Unreachable Code” (000) . . . 72
10.4.10 “Missing Return Value” (000) . . . 73
10.4.11 “Branch Always Taken” (000) . . . 73
10.5 Functions (000) . . . 73
10.5.1 “Reference Not Permitted in Function” (000) . . . 73
10.5.2 “Method Invocation Not Permitted In Function” (000) . . . 73
10.5.3 “Reference Access Not Permitted in Function” (000) . . . 73
10.5.4 “Return from Void” (000) . . . 73
10.6 Types (000) . . . 73
10.6.1 “Subtype Error” (000) . . . 73
10.6.2 “Incomparable Operands” (000) . . . 73
10.6.3 “Record Type Required” (000) . . . 73
10.6.4 “Record Missing Field” (000) . . . 73

Glossary **75**

Chapter 1

Introduction

This document provides a specification of the *Whiley Programming Language*. Whiley is a hybrid imperative and functional programming language designed to produce programs with as few errors as possible. Whiley allows explicit specifications to be given for functions, methods and data structures, and employs a *verifying compiler* to check whether programs meet their specifications. As such, Whiley is ideally suited for use in *safety critical system*s. However, there are many benefits to be gained from using Whiley in a general setting (e.g. improved documentation, maintainability, reliability, etc). Finally, this document is *not* intended as a general introduction to the language, and the reader is referred to alternative documents for learning the language[1].

1.1 Background

Reliability of large software systems is a difficult problem facing software engineering, where subtle errors can have disastrous consequences. Infamous examples include: the Therac-25 disaster where a computer-operated X-ray machine gave lethal doses to patients[2]; the 1988 worm which reeked havoc on the internet by exploiting a buffer overrun[3]; the 1991 Patriot missile failure where a rounding error resulted in the missile catastrophically hitting a barracks[4]; and, the Ariane 5 rocket which exploded shortly after launch because of an integer overflow, costing the ESA an estimated $500 million[5].

Around 2003, Hoare proposed the creation of a *verifying compiler* as a grand challenge for computer science[6]. A verifying compiler *"uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles."* There have been numerous attempts to construct a verifying compiler system, although none has yet made it into the mainstream. Early examples include that of King[7], Deutsch[8], the Gypsy Verification Environment[9] and the Stanford Pascal Verifier[10]. More recently, the Extended Static Checker for Modula-3[11] which became the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work[12]. Building on this success was JML and its associated tooling which provided a standard notation for specifying functions in Java[13]. Finally, Microsoft developed the Spec# system which is built on top of C#[14].

1.2 Goals

The Whiley Programming Language has been designed from scratch in conjunction with a verifying compiler. The intention is to provide an open framework for research in automated software verification. The initial goal is to automatically eliminate common errors, such as *null dereferences*, *array-out-of-bounds*, *divide-by-zero* and more. In the future, the intention is to consider more complex issues, such as termination, proof-carrying code and user-supplied proofs.

1.3 History

Development of the Whiley programming language begun in 2009 by Dr. David J. Pearce, at the time a lecturer in Computer Science at Victoria University of Wellington. The accompanying website `http://whiley.org` went live in 2010, making the first versions of Whiley available for download. Since then, Whiley has been in constant development with the majority of contributions being made by the original author. Several scientific papers have published on different aspects of the language, including:

- **Implementing a Language with Flow-Sensitive and Structural Typing on the JVM**. David J. Pearce and James Noble. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, 2011.
- **Sound and Complete Flow Typing with Unions, Intersections and Negations**, David J. Pearce. In *Proceedings of the Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 335–354, 2013
- **A Calculus for Constraint-Based Flow Typing**. David J. Pearce. In *Proceedings of the Workshop on Formal Techniques for Java-like Languages (FTFJP)*, Article 7, 2013.
- **Whiley: a Platform for Research in Software Verification**. David J. Pearce and Lindsay Groves. In *Proceedings of the Conference on Software Language Engineering (SLE)*, pages 238-âĂŞ248, 2013
- **Reflections on Verifying Software with Whiley**. David J. Pearce and Lindsay Groves. In *Proceedings of the Workshop on Formal Techniques for Safety-Critical Software (FTSCS)*, 2013

Chapter 2

Lexical Structure

This chapter specifies the lexical structure of the Whiley programming language. Programs in Whiley are organised into one or more *source file*s written in Unicode. The Whiley language uses *indentation syntax* to delimit blocks and statements, rather than curly-braces (or similar) as found in many other languages.

2.1 Line Terminators

A Whiley compiler splits the sequence of (Unicode) input characters into lines by identifying *line terminators*:

```
LineTerminator ::= \n | \r | \r \n
```

Here, `\n` represents the ASCII character `LF` (`0xA`), whilst `\r` represents the ASCII character `CR` (`0xD`). The two characters `\r` `\n` taken together form one line terminator.

2.2 Indentation

After splitting the input characters into lines, a Whiley compiler then identifies the *indentation* of each line. This is necessary because Whiley employs indentation syntax meaning that indentation is significant in the meaning of Whiley programs.

```
Indentation ::= ^ ( \t |   )*
```

Here, ˆ demarcates the start of a line and, hence, indentation may only occur at the beginning of a line. Indentation may be compared using the $\leq$ comparator, such that $i \leq ir$ always holds (where i is some indentation and r is either empty or represents additional indentation). In other words, some indentation i is considered less-than-or-equal to another piece of indentation ir which includes the first as a prefix. This comparator is important for delimiting *statement blocks* (§5.1).

2.3 Comments

There are two kinds of comments in Whiley: *line comments* and *block comments*:

```
/* This is a block comment */
```

The above illustrates a block comment, which is all of the text between `/*` and `*/` inclusive.

```
// This is a line comment
```

The above illustrates a line comment, which is all of the text from `//` up to the end-of-line.

2.4 Identifiers

An identifier is a sequence of one or more *letters* or *digits* which starts with a letter.

```
 Ident ::= Letter ( Letter | Digit )*

Letter ::= _ | a | ... | z | A | ... | Z

 Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Letters include lowercase and uppercase alphabetic characters (i.e. `a-z` and `A-Z`) and the underscore (_).

2.5 Keywords

The following strings are reserved for use as *keywords* and may not be used as identifiers:

```
Keyword ::= all | any | assert | assume | bool | break | byte
          | case | catch | continue | debug
          | default | do | else | ensures | export | false
          | fail | finite | for | function | if | import | in | int
          | is | method | native | new | no | null | package
          | private | protected | public | real | requires
          | return | skip | some | switch | throw
          | throws | total | true | try | void | where | while
```

The following strings are reserved for use as *keywords*, but may additionally be used as identifiers in certain contexts:

```
KeywordIdentifier ::= constant | from | type
```

2.6 Literals

A *literal* is a source-level entity which describes a value of primitive type (§4.4).

```
Literal ::= NullLiteral
          | BoolLiteral
          | ByteLiteral
          | IntLiteral
          | RealLiteral
          | CharLiteral
          | StringLiteral
```

2.6.1 Null Literal

The **`null`** type (§4.4.1) has a single value expressed as the **`null`** literal.

```
NullLiteral ::= null
```

2.6.2 Boolean Literals

The **bool** type (§4.4.2) has two values expressed as the `true` and `false` literals.

```
BoolLiteral ::= true | false
```

2.6.3 Byte Literals

The **byte** type (§4.4.3) has 256 values which are expressed as sequences of binary digits, followed by the suffix "`b`" (e.g. `0101b`).

```
ByteLiteral ::= ( 0 | 1 )+ b
```

Byte literals do not need to contain exactly eight digits and, when fewer digits are given, are padded out to eight digits by appending zero's from the left (e.g. `00101b` becomes `00000101b`).

2.6.4 Integer Literals

The **int** type (§4.4.4) represents the infinite set of integer values which are expressed as sequences of numeric or hexadecimal digits (e.g. `123456`, `0xffaf`, etc).

```
IntLiteral ::= ( 0 | ... | 9 )+
             | 0 x ( 0 | ... | 9 | a | ... | f | A | ... | F )+
```

Since integer values in Whiley are of arbitrary size (§4.4.4), there is no limit on the size of an integer literal.

2.6.5 Real Literals

The **real** type (§4.4.5) represents the infinite set of rational values which are expressed as sequences of numeric digits separated by a period (e.g. `1.0`, `0.223`, `12.55`, etc).

$$\texttt{RealLiteral} ::= \left(\boxed{\texttt{0}} \mid \ldots \mid \boxed{\texttt{9}}\right)^+ \boxed{\texttt{.}} \left(\boxed{\texttt{0}} \mid \ldots \mid \boxed{\texttt{9}}\right)^+$$

2.6.6 Character Literals

A *character literal* is expressed as a single character or an escape sequence enclosed in single quotes (e.g. `'c'`).

$$\texttt{CharLiteral} ::= \boxed{\texttt{'}} \left(\texttt{Character} \mid \texttt{Escape}\right) \boxed{\texttt{'}}$$

2.6.7 String Literals

A *string literal* is expressed as a sequence of zero or more characters or escape sequences enclosed in double quotes (e.g. `"Hello␣World"`).

$$\texttt{StringLiteral} ::= \boxed{\texttt{"}} \left(\texttt{Character} \mid \texttt{Escape}\right)^* \boxed{\texttt{"}}$$

Chapter 3

Source Files

Whiley programs are split across one or more source files which are compiled into *WyIL file*s prior to execution. Source files contain declarations which describe the functions, methods, data types and constants which form the program. Source files are grouped together into coherent units called *package*s.

3.1 Compilation Units

Two kinds of *compilation unit* are taken into consideration when compiling a Whiley source file: other source files; and, binary WyIL files. The Whiley Intermediate Language (WyIL) file format is described elsewhere, but defines a binary representation of a Whiley source file. When one or more Whiley source files are compiled together, a *compilation group* is formed. External symbols encountered during compilation are first resolved from the compilation group, and then from previously WyIL files.

3.2 Packages

Programs in Whiley are organised into packages to help reduce name conflicts and provide some grouping of related concepts. A Whiley source file may provide an optional **`package`** declaration to identify the package it belongs to. This declaration must occur at the beginning of the source file.

$$\texttt{PackageDecl} ::= \boxed{\texttt{package}}\ \texttt{Ident} \left(\boxed{\texttt{.}}\ \texttt{Ident} \right)^*$$

Any source file which does not provide a **`package`** declaration is considered to be in the *default package*.

3.3 Names

There are four functional entities which can be defined within a Whiley source file: *type declarations* (§3.5.2), *constant declarations* (§3.5.3), *function declarations* (§3.5.4) and *method declarations* (§3.5.5). These define *named entities* which may be referenced from other compilation units. Every named entity has a unique *fully-qualified* name constructed from the enclosing package name, the source file name and the declared name. For example:

`Graphics.whiley`

```
package tracer

type Point is { int x, int y }

constant Origin is { x: 0, y: 0 }
```

This declares two named entities: `tracer.Graphics.Point` and `tracer.Graphics.Orig`

Two named entities may *clash* if they have the same fully qualified name and are in the same category. There are three entity categories: *types*, *constants* and *functions/methods*. The following illustrates a common pattern:

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

function Point(int x, int y) -> Point:
    return {x: x, y: y}
```

Here, two named entities share the same fully qualified named. This is permitted because they are in different categories.

3.4 Imports

When performing *name resolution*, a Whiley compiler first attempts to resolve names within the same source file. For any remaining unresolved, the compiler examines imported entities in reverse declaration order. Entities are imported using an `import` declaration:

```
ImportDecl ::= import [ FromSpec ] Ident ( ( . | .. ) ( Ident | * ) )*

  FromSpec ::= ( Ident | * ) from
```

A declaration of the form "`import some.pkg.File`" imports the compilation unit "`File`" residing in package "`some.pkg`". Named entities (e.g. "`Entity`") within that compilation unit can then be referenced using a *partially qualified* name which omits the package component (e.g. "`File.Entity`"). A declaration of the form "`import Entity from` imports the named entity "`Entity`" from the compilation unit "`File`" residing in package "`some.pkg`". Note, this does *not* import the compilation unit "`some.pkg.File`" (and, hence, does not subsume the statement "`import some.pkg.File`").

A *wildcard* may be used in place of the compilation unit name (e.g. "`import some.pkg.*`" to import *all* compilation units within the given package. A *package match* may be used in place of some of all of the package component (e.g. "`import some..File`") to import all compilation units with matching name and package *prefix* and/or *suffix*. A *wildcard* may be used in place of the entity name (e.g. "`import * from some.pkg.File`") to import *all* named entities within the given compilation unit.

3.5 Named Declarations

Camel case

3.5.1 Access Control

3.5.2 Type Declarations

A *type declaration* declares a named type within a Whiley source file. The declaration may refer to named types in this or other source files and may also *recursively* refer to itself (either directly or indirectly).

```
TypeDecl ::= type Ident is TypePattern [ where Expr ]
```

The optional **`where`** clause defines a *boolean expression* which holds for any instance of this type. This is often referred to as the type *invariant* or *constraint*. Variables declared within the *type pattern* may be referred to within the optional **`where`** clause.

Examples. Some simple examples illustrating type declarations are:

```
// Define a simple point type
type Point is { int x, int y }

// Define the type of natural numbers
type nat is (int x) where x >= 0
```

The first declaration defines an unconstrained record type named `Point`, whilst the second defines a constrained integer type `nat`.

Notes. A convention is that type declarations for *records* or *unions of records* begin with an upper case character (e.g. `Point` above). All other type declarations begin with lower case. This reflects the fact that records are most commonly used to describe objects in the domain.

3.5.3 Constant Declarations

A *constant declaration* declares a named constant within a Whiley source file. The declaration may refer to named constants in this or other source files, although it may not refer to itself (either directly or indirectly).

```
ConstantDecl ::= constant Ident is Expr
```

The given *constant expression* is evaluated at *compile time* and must produce a constant value. This prohibits the use of function or method calls within the constant expression. However, general operators (e.g. for arithmetic) are permitted.

Examples. Some simple examples to illustrate constant declarations are:

```
// Define the well-known mathematical constant to 10 decimal places.
constant PI is 3.141592654

// Define a constant expression which is twice PI
constant TWO_PI is PI * 2.0
```

The first declaration defines the constant `PI` to have the **`real`** value `3.141592654`. The second declaration illustrates a more interesting constant expression which is evaluated to `6.283185308` at compile time.

Notes. A convention is that constants are named in upper case with underscores separating words (i.e. as in `TWO_PI` above).

3.5.4 Function Declarations

A *function declaration* defines a function within a Whiley source file. Functions are *pure* and may not have side-effects. This means they are guaranteed to return the same result given the same arguments, and are permitted within specifications (i.e. in type invariants, *loop invariant*s, and function/method *precondition*s or *postcondition*s). Functions may call other functions, but may not call other methods. Functions may not allocate memory on the heap and/or instigate concurrent computation.

```
FunctionDecl ::= function Ident TypePattern -> TypePattern (
                   requires Expr | ensures Expr
                 )* : Block
```

The first type pattern (i.e. before "->") is referred to as the *parameter*, whilst the second is referred to as the *return*. There are three kinds of optional clause which follow:

- **Requires clause(s)**. These define constraints on the permissible values of the parameters on entry to the function, and are often collectively referred to as the precondition. These expressions may refer to any variables declared within the parameter type pattern. Multiple clauses may be given, and these are taken together as a conjunction. Furthermore, the convention is to specify the **`requires`** clause(s) before any **`ensures`** clause(s).

- **Ensures clause(s)**. These define constraints on the permissible values of the function's return value, and are often collectively referred to as the postcondition. These expressions may refer to any variables declared within either the parameter or return type pattern. Multiple clauses may be given, and these are taken together as a conjunction. Furthermore, the convention is to specify **`ensures`** clause(s) last.

Examples. The following function declaration provides a small example to illustrate:

```
function max(int x, int y) -> (int z)
// return must be greater than either parameter
ensures x <= z && y <= z
// return must equal one of the parmaeters
ensures x == z || y == z:
    // implementation
    if x > y:
        return x
    else:
        return y
```

This defines the specification and implementation of the well-known `max()` function which returns the largest of its parameters. This does not enforce any preconditions on its parameters.

3.5.5 Method Declarations

A *method declaration* defines a method within a Whiley source file. Methods are *impure* and may have side-effects. Thus, they cannot be used within specifications (i.e.

in type invariants, loop invariants, and function/method preconditions or postconditions). However, unlike functions, they methods call other functions and/or methods (including `native` methods). They may also allocate memory on the heap, and/or instigate concurrent computation.

```
MethodDecl ::= method Ident TypePattern -> TypePattern (
                 requires Expr | ensures Expr
               )* : Block
```

The first type pattern (i.e. before "->") is referred to as the *parameter*, whilst the second is referred to as the *return*. The three optional clauses are defined identically as for function declarations (§3.5.4).

Examples. The following method declaration provides a small example to illustrate:

```
// Define the well-known concept of a linked list
type LinkedList is null | &{ LinkedList next, int data }

// Define a method which inserts a new item onto the end of the list
method insertAfter(LinkedList list, int item) -> LinkedList:
    if list is null:
        // reached the end of the list, so allocate new node
        return new { next: null, data: item }
    else:
        // continue traversing the list
        list->next = insertAfter(list->next, item)
        return list
```

Chapter 4

Types & Values

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. Whiley's *type system* governs how the type of any variable or expression is determined. Whiley's type system is unusual in that it incorporates *union type*s (§4.10), *intersection type*s (§4.11) and *negation type*s (§4.12), as well as employing *flow typing* and *structural typing*.

4.1 Overview

Types in Whiley are unusual (in part) because there is a large gap between their *syntactic* description and their underlying *semantic* meaning. In most programming languages (e.g. Java), this gap is either small or non-existent and, hence, there is little to worry about. However, in Whiley, we must tread carefully to avoid confusion. The following example attempts to illustrate this gap between the syntax and semantics of types:

```
function id(null|int x) -> int|null:
    return x
```

In this function we see two distinct *type descriptor*s expressed in the program text, namely "**int**|**null**" and "**null**|**int**". Type descriptors occur at the source-level and describe *type*s which occur at the abstract (or underlying) level. In this particular case, we have two distinct type descriptors which describe the *same* underlying type. We will often refer to types as providing the semantic (i.e. meaning) of type descriptors.

4.2 Type Descriptors

Type descriptors provide syntax for describing types and, in the remaining sections of this chapter, we explore the range of types supported in Whiley. The top-level grammar for type descriptors is:

```
    Type ::= UnitType ( , UnitType )*

UnitType ::= UnionType
           | IntersectionType
           | TermType

TermType ::=
           | PrimitiveType
           | RecordType
           | ReferenceType
           | NominalType
           | CollectionType
           | NegationType
           | FunctionType
           | MethodType
           | ( Type )
```

4.3 Type Patterns

*Type pattern*s associate variables with types and their subcomponents and can be used to declare variables and/or *destructuring* types into variables. Type patterns are a source-level entity which are similar to type descriptors. The top-level grammar for type patterns is:

```
TypePattern ::= Type [ Ident ]
              | TuplePattern
              | RecordPattern
              | RationalPattern
```

Type patterns do not exist for all compound structures — only those where a value is guaranteed to exist which could be associated with a variable.

4.4 Primitive Types

Primitive types are the atomic building blocks of all types in Whiley.

```
PrimitiveType ::=
               |  AnyType
               |  VoidType
               |  NullType
               |  BoolType
               |  ByteType
               |  IntType
               |  RealType
```

4.4.1 Null

The null type is typically used to show the absence of something. It is distinct from void, since variables can hold the special **`null`** value (where as there is no special "**`void`**" value). The set of values defined by the type **`null`** is the singleton set containing exactly the **`null`** value. Variables of **`null`** type support only equality comparators (§6.6.2.1). The **`null`** value is particularly useful for representing optional values and terminating recursive types.

```
NullType ::= null
```

Example. The following illustrates a simple example of the **`null`** type:

```
type Tree is null | { int data, Tree left, Tree right }

function height(Tree t) -> int:
    if t is null:
        // height of empty tree is zero
        return 0
    else:
        // height is this node plus maximum height of subtrees
        return 1 + Math.max(height(t.left), height(t.right))
```

This defines `Tree` — a *recursive type* — which is either empty (i.e. **`null`**) or consists of a field `data` and two subtrees, `left` and `right`. The `height` function calculates the height of a `Tree` as the longest path from the root through the tree.

Notes. With all of the problems surrounding **`null`** and `NullPointerException`s in languages like Java and C, it may seem that this type should be avoided. However, it remains a very useful abstraction around (e.g. for terminating recursive types) and, in Whiley, is treated in a completely safe manner (unlike e.g. Java).

4.4.2 Booleans

The **`bool`** type represents the set of boolean values (i.e. `true` and `false`). Variables of **`bool`** type support equality comparators (§6.6.2.1), binary logical operators (§6.6.1) and logical not (§**??**).

```
BoolType ::= bool
```

Example. The following illustrates a simple example of the **`bool`** type:

```
// Determine whether item is contained in list or not
function contains([int] list, int item) -> bool:
    // examine every element of list
    int i = 0
    while i < |list|:
        if l == item:
            return true
        i = i + 1
    // done
    return false
```

This function determines whether or not a given integer value is contained within a list of integers. If so, it returns `true`, otherwise it returns `false`.

4.4.3 Bytes

The type **`byte`** represents the set of eight-bit sequences, whose values are expressed numerically using `0` and `1` followed by `b` (e.g. 00101b). The set of values defined by the **`byte`** type is the set of all 256 possible combinations of eight-bit sequences. Variables of **`byte`** type support equality comparators (§6.6.2.1), bitwise operators (§6.6.3), bitwise complement (§**??**) and shift operators (§6.6.7).

```
ByteType ::= byte
```

Example. The following illustrates a simple example of the **byte** type:

```
// convert a byte into a string
function toString(byte b) -> ASCII.string:
    ASCII.string r = "b"
    int i = 0
    while i < 8:
        if (b & 00000001b) == 00000001b:
            r = "1" ++ r
        else:
            r = "0" ++ r
        b = b >> 1
        i = i + 1
    return r
```

This illustrates the conversion from a **byte** into a **string**. The conversion is performed one digit at a time, starting from the rightmost bit.

Notes. Unlike for many languages, there is no representation associated with a byte. For example, to extract an integer value from a byte, it must be explicitly decoded according to some representation (e.g. two's compliment) using an auxillary function (e.g. `Byte.toInt()`).

4.4.4 Integers

The type **int** represents the set of arbitrary-sized integers, whose values are expressed as a sequence of one or more numerical or hexadecimal digits (e.g. `123456`, `0xffaf`, etc). Variables of **int** type support equality comparators (§6.6.2.1), inequality comparators (§6.6.2.2), addition (§**??**), subtraction (§**??**), multiplication (§**??**), division (§**??**), remainder (§**??**) and negation (§6.8.1) operations.

```
IntType ::= int
```

Example. The following illustrates a simple example of the **int** type:

```
function fib(int x) -> int:
    if x <= 1:
        return x
    else:
        return fib(x-1) + fib(x-2)
```

This illustrates the well-known recursive method for computing numbers in the *fibonacci* sequence.

Notes. Since integers in Whiley are of arbitrary size, *integer overflow* is not possible. This contrasts with other languages (e.g. Java) that used *fixed-width* number representations (e.g. 32bit two's complement). Furthermore, there is nothing equivalent to the constants found in such languages for representing the uppermost and least integers expressible (e.g. `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, as found in Java).

4.4.5 Rationals

The type **`real`** represents the set of arbitrary-sized rationals, whose values are expressed as a sequence of one or more numerical digits separated by a period (e.g. `1.0`, `0.223`, `12.55`, etc). The set of values defined by the type **`real`** is the (infinite) set of all integer pairs, where the first element is designated the numerator, and the second designated the denominator. Variables of **`real`** type support equality comparators (§6.6.2.1), inequality comparators (§6.6.2.2), addition (§**??**), subtraction (§**??**), multiplication (§**??**), division (§**??**), remainder (§**??**) and negation (§6.8.1) operations. Variables of type **`real`** also support the *rational destructuring assignment* to extract the numerator and denominator (illustrated below).

```
RealType ::= real
```

Example. The following illustrates a simple example of the **`real`** type:

```
function floor(real x) -> int:
    int num / int den = x     // extract numerator and denominator
    int r = num / den         // integer division
    if x < 0.0 && den != 1:
        return r - 1
    else:
        return r
```

This illustrates the well-known function for computing the *floor* of a **`real`** variable `x` (i.e. the greatest integer not larger than `x`). The rational destructuring assignment is used to extract the numerator and denominator of the parameter `x`.

4.4.6 Any

The type **`any`** represents the type whose variables may hold any possible value. Thus, **`any`** is the *top type* (i.e. $\top$) in the lattice of types and, hence, is the supertype of all other types. Variables of **`any`** type support equality comparators (§6.6.2.1) and *runtime type tests*. Finally, unlike the majority of other types, there are no *values* of type **`any`**.

```
AnyType ::= any
```

Example. The following illustrates a simple example of the **any** type:

```
function toInt(any val) -> int:
    if val is int:
        return val
    else if val is real:
        return Math.floor(val)
    else:
        return 0 // default value
```

Here, the function `toInt` accepts *any valid Whiley value*, which includes all values of type **int**, **real**, collections, records, etc. The function then inspects the value that it has been passed and, in the case of values of type **int** and **real**, returns an integer approximation; for all other values, it returns `0`.

Notes. The any type is roughly comparable to the `Object` type found in pure object-oriented languages. However, in impure object-oriented languages which support primitive types, such as Java, this comparison often falls short because `Object` is not a supertype of primitives such as **int** or `long`.

4.4.7 Void

The **void** type represents the type whose variables cannot exist (i.e. because they cannot hold any possible value). Thus, **void** is the *bottom type* (i.e. $\perp$) in the lattice of types and, hence, is the *subtype* of all other types. Void is used to represent the return type of a method which does not return anything. Furthermore, it is also used to represent the element type of an empty list of set. Finally, unlike the majority of other types, there are no *values* of type **void**.

```
VoidType ::= void
```

Example. The following example illustrates several uses of the **void** type:

```
// Attempt to update first element
method update1st(&[int] list, int value) -> void:
    // First, check whether list is empty or not
    if (*list) != []:
        // Then, update 1st element
```

```
        (*list)[0] = value
    // done
```

Here, the method `update1st` is declared to return **`void`** — meaning it does not return a value. Instead, this method updates some existing state accessible through the reference `list`. Within the method body, the value accessible via this reference is compared against the `[void]` (i.e. the *empty list*).

4.5 Tuples

A tuple type describes a compound type made up of two or more elements in sequence, whose values are expressed as sequences of values separated by a comma (e.g. `1,2`, `2.0,3.32,3.45`, etc). Tuples are similar to records, except that fields are effectively anonymous. Variables of tuple type support equality comparators (§6.6.2.1), as well the *tuple destructuring assignment* to extract elements (illustrated below).

```
TupleType    ::= ( Type ( , Type )+ )
TuplePattern ::= ( TypePattern ( , TypePattern )+ ) [ Ident ]
```

Example. The following example illustrates several uses of tuples:

```
function swap(int x, int y) -> (int,int):
    return y,x
```

This function accepts two integer parameters, and returns a tuple type containing two integers. The function simple reverses the order that the values occur in the tuple passed as a parameter.

4.6 Records

A record type describes a compound made of from one or more *fields*, each of which has a unique name and a corresponding type. Variables of record type support equality comparators (§6.6.2.1) and field access (§6.7) operations, as well as field assignment (§**??**).

```
RecordType ::= { MixedType ( , MixedType )* [ , ... ] }
```

Records use *mixed types* for defining fields (see §**??**), meaning that field names may be mixed within their type. This is primarily useful for fields of function or method type (see below). Records using the `...` notation are referred to as *open records* (e.g. `{int x, ...}`), otherwise they are referred to as *closed records* (e.g. `{int x, int y}`). Open records represent all records containing *at least* the given fields, whilst closed records represent those containing *exactly* the given fields.

Example. The following example illustrates an open record type:

```
type Writer is {
   method write([byte]) -> int,
   ...
}
type PrintWriter is {
   method write([byte]) -> int,
   method println(ASCII.string) -> void,
   ...
}
```

The above illustrates two open records `Writer` and `PrintWriter`. The former has one field (`write`), whilst the latter has two fields (`write` and `println`). The above also illustrates use of mixed types. For example, the field "`write`" is declared as "`method write([byte]) => int`" which mixes together the field name (i.e. "`write`") with its type (i.e. "`method([byte]) => int`").

4.7 References

Reference types in Whiley represent references to variables, such as those allocated in the heap. They are similar to references or pointers found in many imperative and object-oriented languages (e.g. C/C++, Java, C#, etc). A type `&T` represents a reference to a variable of type `T`. Variables of reference type support equality comparators (§6.6.2.1) and dereference (§**??**) operations, as well as dereference assignment (§**??**).

```
ReferenceType ::= & Type
```

Example. The following example illustrates reference types:

```
// Swap contents of heap-allocated int variables
method swap(&int pX, &int pY):
    int tmp = *pX
    *pX = *pY
```

```
    *pY = tmp
```

The above illustrates a method which accepts two references to variables of type **int** that may refer to the same variable. The method simply swaps the contents of the variables to which they refer.

4.8 Collections

Collection types in Whiley describe compound values constructed from arbitrarily many values.

```
CollectionType ::= SetType
                 | MapType
                 | ListType
```

4.8.1 Sets

A set type describes set values whose elements are subtypes of the element type. For example, `{1,2,3}` is an instance of set type `{int}`; however, `{1.345}` is not. Variables of set type support equality comparators (§6.6.2.1), union (§**??**), intersection (§**??**), difference (§**??**) and element-of (§**??**) operations.

```
SetType ::= { Type }
```

Example. The following example illustrates set types:

```
// Adjacency list representation
type Graph is ([{int}])

function depthFirstSearch(int v, Graph graph, {int} visited) -> {int}:
    visited = visited + {v}
    // Traverse edges not yet visited
    for w in graph[v]:
        if !(w in visited):
            visited = depthFirstSearch(w,graph,visited)
    // Done
    return visited
```

The above illustrates a simple implementation of the well-known *depth-first search* algorithm. In the example, a `Graph` is a list of sets of edge targets, where any `w in g[v]` describes an edge from `v` to `w` in the graph. The `visited` set is used to maintain a list of previously seen vertices, in order to prevent the same vertex from being visited more than once.

4.8.2 Maps

A map represents a one-many mapping from variables of one type to variables of another type. For example, the map type `{int=>real}` represents a map from integers to real values. A valid instance of this type might be `{1=>1.2,2=>3.0}`. Variables of map type support equality comparators (§6.6.2.1), access (§6.7), union (§**??**), intersection (§**??**), difference (§**??**) and element-of (§**??**) operations.

```
MapType ::= { Type => Type }
```

Example. The following example illustrates map types:

```
type Expr is int | string // simple expression forms

function evaluate(Expr e, {string=>int} environment) -> int:
    if e is int:
        // expression is constant, so return this directly
        return e
    else:
        // expression is variable, so look up its value in environment
        return environment[e]
```

The above illustrates a function for evaluating simple expressions which are either integer constants or variable names. To evaluate an expression which is an integer constant, we simply return that constant. To evaluate an expression which is a variable name, we look up the current value of that variable in an environment which maps variable names to integer constants.

4.8.3 Lists

A list type describes list values whose elements are subtypes of the element type. For example, `[1,2,3]` is an instance of list type `[int]`; however, `[1.345]` is not. Variables of list type support equality comparators (§6.6.2.1), append (§6.6.5), sublist (§**??**) and element-of (§**??**) operations.

```
ListType ::= [ Type ]
```

Example. The following example illustrates list types:

```
function add([int] v1, [int] v2) -> ([int] v3):
    int i=0
    while i < |v1|:
        v1[i] = v1[i] + v2[i]
        i = i + 1
    return v1
```

The above illustrates a simple function which adds two integer lists together. The function's precondition requires that both input list have the same length, whilst its postconditions ensures that this matches the length of the output.

4.9 Functions and Methods

A function or method type describes the signature of a function or method. These types enable functions or methods to be passed around as values in Whiley and are often referred to as *functors*. This enables a degree of polymorphism in the language, where the exact function or method to be called is unknown. Variables of function or method type support equality comparators (§6.6.2.1) only.

```
FunctionType ::= function ( [ Type ( , Type )* ] ) => UnitType

  MethodType ::= method ( [ Type ( , Type )* ] ) => UnitType
```

Example. The following example illustrates function types:

```
type Fun is function(int) -> int

function map([int] items, Fun fn) -> [int]:
    //
    int i = 0
    while i < |items|:
        items[i] = fn(items[i])
        i = i + 1
    //
    return items
```

The above illustrates the well-known *map* function, which maps all elements of a list according to a given function.

4.10 Unions

A union type is constructed from two or more component types and contains any value held in any of its components. For example, the type `null|int` is a union which holds either an integer value or `null`. The set of values defined by a union type `T1|T2` is exactly the union of the sets defined by `T1` and `T2`. In general, variables of union type support only equality (§**??**), inequality comparisons (§**??**) and *runtime type tests* (see §4.14 for exceptions to this).

```
UnionType ::= IntersectionType ( | IntersectionType )*
```

Example. The following example illustrates a union type:

```
// Return lowest index of matching item, or null if none
function indexOf([int] items, int value) -> int|null:
    int i = 0
    while i < |items|:
        if items[i] == value:
            // match
            return i
        i = i + 1
    // item not found
    return null
```

Here, a union type is used to construct a more expressive return value. If no matching element is found, `null` is returned (rather than e.g. `-1`).

4.11 Intersections

An intersection type is constructed from two or more component types and contains any value held in all of its components. For example, the type `[int]&[bool]` is an intersection which hold any value which is both an instance of `[int]` and `[bool]` (in fact, only the empty list meets this criteria). Intersections are used to type variables on the true branch of a runtime type test. The set of values defined by an intersection type `T1&T2` is exactly the intersection of the sets defined by `T1` and `T2`. In general, variables of intersection type support only equality (§**??**), inequality comparisons (§**??**) and *runtime type tests* (see §4.14 for exceptions to this).

```
IntersectionType ::= TermType ( & TermType )*
```

Example. The following example illustrates an intersection type:

```
type Reader is {
   method read(int) -> [byte],
   ...
}
type Writer is {
   method write([byte]) -> int,
   ...
}
type ReaderWriter is Reader & Writer
```

Here, the type `Reader` is defined as any record containing a `read(int)` method, whilst the type `Writer` is defined as any record containing a `write([byte])` method. Then, the intersection type `ReaderWriter` is defined as any record containing *both* a `read(int)` and `write([byte])` method.

4.12 Negations

A negation type is constructed a component type and contains any value *not* held in its component. For example, the type `!int` is a negation which holds any non-integer value. Negations are used to type variables on the false branch of a runtime type test. The set of values defined by a negation type $!\mathtt{T}_1$ is exactly the set of all values less those defined by $\mathtt{T}_1$. In general, variables of negation type support only equality (§**??**), inequality comparisons (§**??**) and *runtime type tests* (see §4.14 for exceptions to this).

```
NegationType ::= ! TermType
```

Example. The following example illustrates a negation type:

```
function f(any item) -> !null:
    if item is null:
        return 0
    else:
        return item
```

Here, the function `f()` accepts a parameter of any type, and returns a value which is permitted to be anything except `null`. The above also illustrates how the type test operator (§**??**) retypes variables on the false branch using negation types.

4.13 Recursive Types

Recursive types describe tree-like structures of arbitrary depth. For example, linked lists, binary trees, quad trees, etc can all be described using recursive types. Recursive types have no explicit syntax and, instead, are declared indirectly in terms of themselves using one or more nominal types (§??).

Example. The following example illustrates a simple recursive type:

```
type Node is { Tree left, Tree right, int data }
type Tree is null | Node

function sizeOf(Tree t) -> int:
    if t == null:
        return 0
    else:
        return 1 + sizeOf(t.left) + sizeOf(t.right)
```

Here, the type `Tree` is recursive because it is defined in terms of itself. An instance of type `Tree` is a sequence of nested records which is arbitrarily deep, and whose branches are terminated by `null`. The function `sizeOf()` traverses an arbitrary instance of `Tree` and returns the number of `Node`s it contains.

4.14 Effective Types

An effective type is a union of types which all contain some property (e.g. a union of lists). This common property allows the effective type to support more operations than possible for an arbitrary union (§4.10).

4.14.1 Effective Tuples

An effective tuple is a union of tuple types. For example, `(int,int)|(real,real)` is an effective tuple. An effective tuple type supports all operations valid for a tuple type (§4.5).

4.14.2 Effective Records

An effective record is a union of record types. For example, `{int f, int g}|{real f, int` is an effective record. An effective record provides access to fields common to all records in the union. For example, the type `{int f, int g}|{real f, int h}` can be viewed as having an effective type of `{int|real f, ...}` and, hence, read access to field `f` is given.

4.14.3 Effective Collections

An effective collection is a union of collection types. For example, `[int]|[real]` is an effective list. An effective collection supports all operations valid for a collection type (§4.8). For example, the type `[int]|[real]` can be viewed as having an effective type of `[int|real]` and, hence, read access to its length and elements is given.

4.15 Semantics

Although types are abstract entities we can (for the most part) imagine them as describing sets of *abstract values*. For example, `int|null` denotes the set of values containing exactly the (infinite) set of integers and `null` (i.e. $\mathbb{Z} \cup \{\texttt{null}\}$). This is often referred to as a set-theoretic interpretation of types[15;16;17;18]. Under this interpretation, for example, one type *subtypes* another if the set of values it denotes is a *subset* of the other (see § 4.15.2 for more).

4.15.1 Equivalences

Since types are defined in terms of the set of values they represent, it is possible for two distinct type descriptors to describe the same underlying type. For example, `int|null` is considered equivalent to `null|int`. Whilst this case is fairly easy to spot, others are not so obvious. Some examples are given here to illustrate:

- `!any` is equivalent to `void` and, conversely, `any` is equivalent to `!void`
- `int&!int` is equivalent to `void` and, conversely, `int|!int` is equivalent to `any`
- `{int|null f}` is equivalent to `{int f}|{null f}`
- `{int|null f}&{bool|null f}` is equivalent to `{null f}`

Unfortunately, an infinite number of equivalences exist between the type descriptors of Whiley, and we cannot list them all here.

4.15.2 Subtyping

Types in Whiley support the notion of *subtyping* where one type may be a *subtype* for another. For example, the type `int` is a subtype of `any`. Likewise, `bool` is a subtype of `bool|null`. The *subtyping operator* is denoted by "$\leq$"; for example, $\texttt{T}_1 \leq \texttt{T}_2$ indicates that type $\texttt{T}_1$ is a subtype of $\texttt{T}_2$. The subtyping operator is *reflexive*, *transitive* and *antisymmetric* with respect to the underlying types involved.

The subtyping operator is regarded as an algorithm for determining whether the type described by one type descriptor is a subtype of another. The implementation of this algorithm is not straightforward and a full discussion of it is beyond the scope of this document. Indeed, there are many possible implementations of this operator.

Chapter 5

Statements

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

5.1 Blocks

A statement block is a sequence of zero or more consecutive statements which have the same indentation. Statement blocks are used to group statements together when constructing compound statements. For example:

```
function sum([int] items) -> int:
    // outer block begins
    int r = 0
    int i = 0
    while i < |items|:
        // inner block begins
        r = r + items[i]
        i = i + 1
        // inner block ends
    //
    return r
    // outer block ends
```

The above example contains two statement blocks, one nested inside the other. The outer block demarcates the body of the `sum()` function, whilst the inner block demarcates the body of the **while** statement.

5.2 Assert Statement

Represents an *assert statement* of the form "**assert** e", where e is a boolean expression. A *fault* will be raised at runtime if the asserted expression evaluates to `false`; otherwise, execution will proceed normally. At verification time, the verifier is forced to ensure that the asserted expression is true for all possible execution paths. This allows the programmer to specify and check something he/she believes to be true at a given point in the program.

```
AssertStmt ::= assert Expr
```

Example. The following illustrates an **assert** statement:

```
function abs(int x) -> int:
    if x < 0:
        x = -x
    assert x >= 0
    return x
```

Here, an assertion is used to check that the value being returned by the `abs()` is non-negative. Since this is a true statement of the function, this statement will never raise a fault.

5.3 Assignment Statement

As *assignment statement* is of the form `leftHandSide = rightHandSide`. Here, the `rightHandSide` is any expression, whilst the `leftHandSide` must be an `LVal` — that is, an expression permitted on the left-hand side of an assignment. At runtime, the value generated by evaluating the right-hand side must be a subtype (§4.15.2) of the left-hand side.

```
AssignStmt ::= LVal = Expr
```

Example. The following illustrates different possible assignment statements:

```
method f1([int] x, [int] y):
    x = y          // variable assignment

method f1({int f} x, int y):
```

```
    x.f = y        // field assignment

method f1([int] x, int i, int y):
    x[i] = y       // list assignment

method f1([{int f}] x, int i, int y):
    x[i].f = y    // compound assignment
```

The last assignment here illustrates that the left-hand side of an assignment can be arbitrarily complex, involving nested assignments into lists and records.

5.4 Assume Statement

An *assume statement* is of the form "`assume e`", where `e` is a boolean expression. A fault will be raised at runtime if the assumed expression evaluates to `false`; otherwise, execution will proceed normally. At verification time, the verifier will automatically assume that the given expression holds. Thus, `assume` statements provide a way for the programmer to override the verifier. This is useful where the verifier is unable to establish something that the programmer knows to be true. Care must be taken to ensure that the assumed expression really does hold.

```
AssumeStmt ::= assume Expr
```

Example. The following illustrates an `assume` statement:

```
function abs(int x) -> (int y) ensures y >= 0:
    //
    assume x >= 0
    return x
```

Here, the programmer has used an assumption to ensure this function passes verification. This would not appear to be safe in this case, and may lead to a fault at runtime.

5.5 Break Statement

A *break statement* transfers control out of the enclosing loop (i.e. **`do`**, **`for`**, **`while`**). It is a compile-time error if no such enclosing loop exists.

```
BreakStmt ::= break
```

Example. The following illustrates a **`break`** statement:

```
// Remove lowest element holding x from xs
function remove([int] xs, int x) -> [int]:
    int i = 0
    while i < |xs|:
        if xs[i] == x:
           break
        else:
           i = i + 1
    return xs[0..i] ++ xs[i+1..]
```

Here, we see a **`break`** statement being used to exit a **`while`** loop when the first element matching parameter `x` is found.

Notes. Unlike many other programming languages (e.g. Java), **`break`** statements cannot be used to transfer control out of a **`switch`** statement (§5.15). This is because **`switch`** statements have *explicit*, rather than *implicit*, fall-through.

5.6 Continue Statement

A *continue statement* can be used either to transfer control to the next iteration of the enclosing loop (i.e. **`do`**, **`for`**, **`while`**), or to transfer control to the next case of the enclosing **`switch`** statement.

```
ContinueStmt ::= continue
```

Example. The following illustrates a **`continue`** statement:

```
function sumNonNegative([int] xs) -> int:
    int i = 0
    int r = 0
    while i < |xs|:
        if xs[i] < 0:
            continue
        r = r + xs[i]
        i = i + 1
```

```
    return r
```

Here, a **continue** statement is used to ensure the negative numbers are not included in the result of the function.

Notes. Unlike many other programming languages (e.g. Java), **continue** statements are used to transfer control to the next case of a **switch** statement (§5.15). This is because **switch** statements have *explicit*, rather than *implicit*, fall-through.

5.7 Debug Statement

A *debug statement* outputs the result of evaluating its expression to the *debug stream*. Debug statements are intended to be used purely for debugging, particularly from within (pure) functions. The debug stream is an imaginary output stream which does not exist in the true semantic of the language. Instead, from an operational semantics perspective, the debug statement is equivalent to the skip statement (§5.14).

```
DebugStmt ::= debug Expr
```

Example. The following illustrates a `debug` statement:

```
function f(int x) -> int:
    debug "f(int)␣called"
    if x == 1 || x == 0:
       return x
    else:
       return f(x-1) + f(x-2)
```

Here, we see a recursive implementation of the well-known *fibonacci* sequence. A `debug` statement is being used to investigate the parameter values passed to the function.

5.8 Do/While Statement

A do-while statement repeatedly executes a statement block until an expression (the condition) evaluates to `false`. Optional **where** clause(s) are permitted which, together, are commonly referred to as the loop invariant.

DoWhileStmt$^\ell$ ::= `do` `:` Block$^\gamma$ `while` Expr (`where` Expr)*

(where $\ell < \gamma$)

Example. The following illustrates an do-while statement:

```
function sum([int] xs) -> int
// Input must not be empty list
requires |xs| > 0:
  //
  int r = 0
  int i = 0
  do:
    r = r + xs[i]
    i = i + 1
  while i < |xs| where i >= 0
  //
  return r
```

Here, we see a simple do-while statement which sums the elements of variable `xs`, storing the result in variable `r`. A loop invariant is given which establishes that variable `i` is non-negative.

Notes. When multiple **`where`** clauses are given, these are combined using a conjunction to form the loop invariant. The combined invariant must hold on entry to the loop and after each iteration. Thus, when the condition evaluates to `false`, the loop invariant is guaranteed to hold. However, the loop invariant need not hold when the loop is exited using a **`break`** (§5.5) statement.

5.9 For Statement

A *for statement* iterates over all elements in a collection obtained from evaluating the *source expression*. Optional **`where`** clause(s) are permitted which, together, are commonly referred to as the loop invariant.

ForStmt$^\ell$ ::= `for` VarPattern `in` Expr (`where` Expr)* `:` Block$^\gamma$

(where $\ell < \gamma$)

Example. The following illustrates a **for** statement:

```
function max([int] items) -> int
// Input list cannot be empty
requires |items| > 0:
   //
   int r = items[0]

   for v in items:
        r = Math.max(r,v)

   return r
```

Here, we see a simple **for** loop which iterates over all elements of the list `items`. At each iteration, variable `i` holds the index whilst `v` contains the element at that index (i.e. `v == items[i]`).

Notes. When multiple **where** clauses are given, these are combined using a conjunction to form the loop invariant. The combined invariant must hold on entry to the loop and after each iteration. Thus, when the condition evaluates to `false`, the loop invariant is guaranteed to hold. However, the loop invariant need not hold when the loop is exited using a **break** (§5.5) statement.

5.10 Fail Statement

A *fail statement* is used to signal unreachable code. At runtime, this forces abrupt termination of the program. At verification time, the verifier will ensure the statement is unreachable.

```
FailStmt ::= fail
```

Example. The following illustrates a `fail` statement:

```
type nat is (int x) where x >= 0
type neg is (int x) where x < 0

function f(int|null x) -> bool|null:
  //
  if x is nat:
     return true
  else if x is neg:
     return false
```

```
    else:
        fail
```

Here, we see a simple function which checks whether its parameter `x` is positive or negative. A `fail` statement is used to signal that the last branch is, in fact, unreachable.

5.11 If Statement

An **if** statement conditionally executes a statement block based on the outcome of one or more expressions. Chaining of **if** statements is permitted, and an optional **else** branch may be given. The expression(s) are referred to as *conditions* and must be boolean expressions. The first block is referred to as the *true branch*, whilst the optional **else** block is referred to as the *false branch*.

$$\texttt{IfStmt}^{\ell} ::= \boxed{\texttt{if}}\ \texttt{Expr}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma} \left(\boxed{\texttt{else}}\ \boxed{\texttt{if}}\ \texttt{Expr}\ \boxed{\texttt{:}}\ \texttt{Block}^{\omega_i} \right)^{*} \left[\boxed{\texttt{else}}\ \boxed{\texttt{:}}\ \texttt{Block}^{\phi} \right]$$

(where $\ell < \gamma$ and $\forall i.\ell < \omega_i$ and $\ell < \phi$)

Example. The following illustrates an **if** statement:

```
function max(int x, int y) -> int:
    if(x > y):
        return x
    else if(x == y):
        return 0
    else:
        return y
```

Here, we see an **if** statement with two conditional outcomes and one default outcome.

5.12 While Statement

A while statement repeatedly executes a statement block until an expression (the condition) evaluates to `false`. Optional **where** clause(s) are permitted which, together, are commonly referred to as the loop invariant.

$\texttt{WhileStmt}^{\ell} ::= \boxed{\texttt{while}}\ \texttt{Expr}\ \left(\boxed{\texttt{where}}\ \texttt{Expr}\right)^{*}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}$

(where $\ell < \gamma$)

Example. The following illustrates an **`while`** statement:

```
function sum([int] xs) -> int:
  int r = 0
  int i = 0
  while i < |xs| where i >= 0:
    r = r + xs[i]
    i = i + 1
  return r
```

Here, we see a simple **`while`** statement which sums the elements of variable `xs`, storing the result in variable `r`. A loop invariant is given which establishes that variable `i` is non-negative.

Notes. When multiple **`where`** clauses are given, these are combined using a conjunction to form the loop invariant. The combined invariant must hold on entry to the loop and after each iteration. Thus, when the condition evaluates to `false`, the loop invariant is guaranteed to hold. However, the loop invariant need not hold when the loop is exited using a **`break`** (§5.5) statement.

5.13 Return Statement

A *return statement* has an optional expression referred to as the *return value*. At runtime, this statement returns control to the caller of the enclosing function or method. At verification time, the verifier will ensure the returned value meets the postcondition of the enclosing function or method.

$\texttt{ReturnStmt} ::= \boxed{\texttt{return}}\ \left[\texttt{Expr}\right]$

Example. The following illustrates a **`return`** statement:

```
function f(int x) -> int:
    return x + 1
```

Here, we see a simple simple function which returns the increment of its parameter `x` using a **`return`** statement.

Notes. The returned expression (if there is one) must begin on the same line as the statement itself.

5.14 Skip Statement

A *skip statement* is a no-operation and has no effect on the environment. This statement can be useful for representing empty statement blocks (§5.1).

```
SkipStmt ::= skip
```

Example. The following illustrates a **skip** statement:

```
function abs(int x) -> (int y)
// Return value cannot be negative
ensures y >= 0:
    //
    if x >= 0:
        skip
    else:
        x = -x
    //
    return x
```

Here, we see a **skip** statement being used to represent an empty statement block.

5.15 Switch Statement

A *switch statement* transfers control to one of several statement blocks, referred to as *switch cases*, depending on the value obtained from evaluating a given expresion. Each case is associated with one or more values which are used to match against. If no match is made, control either falls through to the next statement following the **switch** or is transferred to a **default** block if one is given.

$$\texttt{SwitchStmt}^{\ell} ::= \boxed{\texttt{switch}}\ \texttt{Expr}\ \boxed{\texttt{:}}\ \Big(\texttt{CaseBlock}^{\gamma} \mid \texttt{DefaultBlock}^{\gamma}\Big)^{+}$$

$$\texttt{CaseBlock}^{\ell} ::= \boxed{\texttt{case}}\ \texttt{ConstantExpr}\ \Big(\boxed{\texttt{,}}\ \texttt{ConstantExpr}\Big)^{*}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}$$

$$\texttt{DefaultBlock}^{\ell} ::= \boxed{\texttt{default}}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma}$$

(where $\ell < \gamma$)

Example. The following illustrates a **`switch`** statement:

```
function toDescriptorString(Primitive t) -> string:
    switch t:
        case Boolean:
            return "Z"
        case Byte:
            return "B"
        case Char:
            return "C"
        case Short:
            return "S"
        case Int:
            return "I"
        case Long:
            return "J"
        case Float:
            return "F"
        default:
            return "D"
```

Here, we see a simple **`switch`** statement which choose between a number of possible values of type `Primitive`. A **`default`** case is given which catches the only remaining case (i.e. representing the value `Double`).

5.16 Variable Declaration Statement

A *variable declaration* statement has an optional expression assignment referred to as a *variable initialiser*. If an initialiser is given, this will be evaluated and assigned to the declared variables when the declaration is executed.

```
VarDecl ::= TypePattern [ = Expr ]
```

Example. Some example variable declarations are:

```
method f():
    int x
    int y = 1
    int z = y + y
    int a, int b = y,z
```

Here we see four variable declarations. The first has no initialiser, whilst the remainder have initialisers. The final declaration illustrates a more complex use of type patterns where two variables of type **`int`** are initialised from a tuple expression

Chapter 6

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.

6.1 Evaluation Order

6.2 Multi Expressions

A multi-expression is an expression composed of two or more unit expressions and which returns a tuple value (§4.5). The operands of a multi-expression are evaluated in a strict left-to-right order.

```
TupleExpr ::= UnitExpr ( , UnitExpr )+
```

Example. The following example illustrates the use of a multi-expression:

This function accepts three **`real`** values and returns two. In the body, the **`return`** statement contains a multi-expression which produces a tuple composed of two **`real`** values.

6.3 Unit Expressions

A unit expression is an expression which returns exactly one value. There is a large range of possible unit expressions, including comparators, arithmetic operators, logical operators, etc.

```
UnitExpr ::= LogicalExpr
           | BitwiseExpr
           | ConditionExpr
           | QuantifierExpr
           | AppendExpr
           | RangeExpr
           | ShiftExpr
           | AdditiveExpr
           | MultiplicativeExpr
           | AccessExpr
           | UnaryExpr
           | LambdaExpr
           | TermExpr
```

6.4 Logical Expressions

A logical expression operates on values of **`bool`** type (§4.4.2) to produce another **`bool`** value. The *if-and-only-if (iff)* operator, `<==>`, returns `true` if either both operands are `true` or both are `false`. The *implication* operator, `==>`, returns `true` if either the left operand is `false`, or both operands are `true`. The *logical OR* operator returns `true` if either operand is `true`, whilst the *logical AND* operator returns `true` if both operands are `true`.

```
   LogicalExpr ::= LogicalOrExpr [ <==> LogicalExpr ]
                 | LogicalOrExpr [ ==> LogicalExpr ]

 LogicalOrExpr ::= LogicalAndExpr
                 | LogicalOrExpr || LogicalAndExpr

LogicalAndExpr ::= BitwiseExpr
                 | LogicalAndExpr && BitwiseExpr
```

Example. The following examples illustrate some of the logical operators:

The function `implies()` implements the well-known equivalence between implication and logical OR. The function `iff()` implements the well-known equivalence between implication and iff.

6.5 Quantifier Expressions

```
QuantExpr ::= ( no | some | all ) {
              Ident in Expr ( , Ident in Expr )+ | LogicalExpr
              }
```

Description.

Examples.

Notes.

6.6 Binary Expressions

6.6.1 Logical Expressions

A logical expression operates on values of **`bool`** type (§4.4.2) to produce another **`bool`** value. The *if-and-only-if (iff)* operator, `<==>`, returns `true` if either both operands are `true` or both are `false`. The *implication* operator, `==>`, returns `true` if either the left operand is `false`, or both operands are `true`. The *logical OR* operator returns `true` if either operand is `true`, whilst the *logical AND* operator returns `true` if both operands are `true`.

```
   LogicalExpr ::= LogicalOrExpr [ <==> LogicalExpr ]
                 | LogicalOrExpr [ ==> LogicalExpr ]

 LogicalOrExpr ::= LogicalAndExpr
                 | LogicalOrExpr || LogicalAndExpr

LogicalAndExpr ::= BitwiseExpr
                 | LogicalAndExpr && BitwiseExpr
```

Example. The following examples illustrate some of the logical operators:

The function `implies()` implements the well-known equivalence between implication and logical OR. The function `iff()` implements the well-known equivalence between implication and iff.

6.6.2 Relational Expressions

A relational expression compares two (or more) values together producing a value of **`bool`** type. Such expressions determine whether a given relationship exists between values.

6.6.2.1 Equality Expressions

The *equality comparator*, ==, tests whether two values are equal. Likewise, the *inequality comparator*, ==, tests whether two values are *not* equal.

```
ConditionExpr ::=
                | AppendExpr == AppendExpr
                | AppendExpr != AppendExpr
```

Example. The following example illustrates an equality expression:

This function checks whether a given integer is contained in a list of integers. This is done by iterating each element of the list and comparing it against the given item.

6.6.2.2 Inequality Expressions

Inequality expressions are either *strict* (where only inequality is tested) or *non-strict* (where both equality and inequality are tested). The *less-than comparator*, <, and *greater-than comparator*, >, are strict. Conversely, the *less-than-or-equal comparator*, <=, and *greater-than-or-equal comparator*, >, are non-strict.

```
ConditionExpr ::=
                | AppendExpr < AppendExpr
                | AppendExpr <= AppendExpr
                | AppendExpr => AppendExpr
                | AppendExpr > AppendExpr
```

Example. The following example illustrates the strict inequality comparators:

This function compares two integer arguments and returns the "sign" of their comparison. The strict inequality comparators are used so the case where `x == y` can be distinguished.

6.6.2.3 Subset Expressions

Subset expressions are either *strict* (where only inequality is tested) or *non-strict* (where both equality and inequality are tested). The *subset comparator*, ⊂, is strict. Conversely, the *subset-or-equal comparator*, ⊆, is non-strict

```
ConditionExpr ::=
             | AppendExpr ⊂ AppendExpr
             | AppendExpr ⊆ AppendExpr
```

Example. The following example illustrates the strict comparator:

This function compares two integer sets and returns the "sign" of their comparison. The non-strict inequality comparators are used so the cases where `xs == ys` and `xs != ys` can be distinguished.

Notes. At the time of writing there is no keyword or non-unicode operator for expressing these comparators.

6.6.2.4 Element-Of Expression

An element-of expression determines whether one value is contained within another value of collection type (§4.8) and produces a value of **`bool`** type. Specifically, if the element is contained then the result is `true`, otherwise it is `false`.

```
ConditionExpr ::=
             | AppendExpr in AppendExpr
             | AppendExpr ∈ AppendExpr
```

Example. The following illustrates the element-of operator.

This function accepts two integer sets and checks whether any element of the first parameter is in the second.

6.6.3 Bitwise Expressions

A bitwise expression operates on values of **`byte`** type (§4.4.3). The *bitwise OR* operator, `|`, performs a logical OR between the respective bits of each operand, and produces a **`byte`**. The *bitwise AND* operator, `&`, performs a logical AND between the respective bits of each operand, and produces a **`byte`**. The *bitwise exclusive-OR* operator, `^`, performs a logical exclusive-OR between the respective bits of each operand, and produces a **byte**.

```
    BitwiseExpr ::= BitwiseOrExpr

  BitwiseOrExpr ::= BitwiseXorExpr
                  | BitwiseOrExpr | BitwiseXorExpr

 BitwiseXorExpr ::= BitwiseAndExpr
                  | BitwiseXorExpr ^ BitwiseAndExpr

 BitwiseAndExpr ::= ConditionExpr
                  | BitwiseAndExpr && ConditionExpr
```

Example. The following example illustrates the bitwise OR operator:

```
function toUnsignedByte(u8 v) => byte:
    //
    byte mask = 00000001b
    byte r = 0b
    for i in 0..8:
        if (v % 2) == 1:
            r = r | mask
        v = v / 2
        mask = mask << 1
    return r
```

This function converts an unsigned integer in the range `0` ... `255` to a **byte**. The bitwise OR operator is used to construct the resulting **byte** by setting individual bits via the `mask`. This example also illustrates the left-shift operator (§6.6.7).

6.6.4 Set Expressions

Set union, intersection and difference.

6.6.5 Append Expressions

An append expression accepts two lists arguments and produces a value of list type. The append operator, ++, concatenates the argument lists together, producing a list constructed from those elements of the first argument (in order0 followed by those of the second argument (in order).

```
AppendExpr ::= RangeExpr ( ++ RangeExpr )*
```

Examples. The following example illustrates the list append operator:

This function constructs a list containing the integer values from `start` up to (but not including) `end`. The list append operator is used to construct the list, starting initially from the empty list.

6.6.6 Range Expressions

A range expression accepts two integer arguments and produces a value of integer list type. The *range operator* constructs the list of integer values from the first argument up to (but not including) the last.

```
RangeExpr ::= ShiftExpr [ .. ShiftExpr ]
```

Examples. The following illustrates the range operator:

This function sums the integers between `start` and `end`. The range operator is used to construct the list to be iterated over.

Notes. If the start of a range expression is greater than its end, the empty list is returned.

6.6.7 Shift Expressions

A shift expression accepts an argument of **`byte`** type (left) and one of **`int`** type (right) and produces a value of **`byte`** type. The *left shift operator*, `<<`, shifts the bits of a **`byte`** in an upwards direction, such that the most significant bit is discarded and the least significant bit assigned `0`. The *right shift operator*, `>>`, shifts bits in a downwards direction, such that the least significant bit is discarded and the most significant bit assigned `0`.

```
ShiftExpr ::= AdditiveExpr [ ( « | » ) AdditiveExpr ]
```

Examples. The following illustrates the left shift operator:

This function accepts an integer between `0` and `255` and converts this into an appropriate bit representation. The left shift operator is used to maintain an internal mask for the bit currently being initialised.

6.6.8 Additive Expressions

An additive expression accepts two arguments of identical type (either **`int`** or **`real`**) and produces a result of matching type. The *addition operator*, `+`, adds both arguments together whilst the *subtraction operator*, `-`, subtracts its right argument from its left argument.

```
AdditiveExpr ::= MultiplicativeExpr [ ( + | - ) AdditiveExpr ]
```

Example. The following illustrates the additive operators:

This function simply computes the difference between its two arguments using the subtraction operator.

6.6.9 Multiplicative Expressions

A multiplicative expression accepts two arguments of identical type (either **`int`** or **`real`**) and produces a result of matching type. The *multiplication operator*, `*`, multiplies both arguments together whilst the *division operator*, `/`, divides its left argument by its right argument. Finally, the *remainder operator* returns the remainder of its operands from an implied division.

```
MultiplicativeExpr ::= AccessExpr [ ( * | / | % ) MultiplicativeExpr ]
```

Example. The following illustrates the remainder operator:

This function accepts a non-negative integer and uses this to index into a list. To ensure the list access is within bounds, the remainder operator is used. Furthermore, the function requires the list is non-empty to prevent a fault with the remainder operator.

Notes. For division, the right operator must be non-zero otherwise a fault is raised, and likewise for remainder. For integer division, the result is rounded towards zero. For a remainder operation, the result may be negative (e.g. `-4 % 3 == -1`).

6.7 Access Expressions

```
AccessExpr ::= TermExpr
               ( [ AdditiveExpr ]
               | [ AdditiveExpr .. AdditiveExpr ]
               | . Ident [ ( ArgsList ) ]
               | -> Ident [ ( ArgsList ) ]
               )*

  ArgsList ::= [ UnitExpr ( , UnitExpr )* ]
```

Description. The *arrow operator* returns a field of the value referenced by the argument.

Examples.

Notes. The arrow operation "`e->f`" is a short-hand notation for "`(*e).f`" and can be used when `e` has effective record type (§4.14.2).

6.8 Unary Expressions

```
UnaryExpr ::= NegationExpr
            | LengthOfExpr
            | BracketedExpr
            | CastExpr
            | LogicalNotExpr
            | BitwiseCompExpr
            | NewExpr
            | DereferenceExpr
```

6.8.1 Negation Expression

A negation expression accepts one argument of numeric type and produces a result of matching type. Specifically, the *negation operator* mathematically negates the given value, which is always equivalent to subtracting the operand from zero.

```
NegationExpr ::= - AccessExpr
```

Example. The following illustrates the negation operator:

Notes.

6.8.2 LengthOf Expression

```
LengthOfExpr ::= | AppendExpr |
```

Example.

Notes.

6.8.3 Bracketed Expression

```
BracketedExpr ::= ( MultiExpr )
```

Example.

Notes.

6.8.4 Cast Expression

```
CastExpr ::= ( DefiniteType ) MultiExpr
```

Example.

Notes.

6.8.5 Logical Not Expression

```
LogicalNotExpr ::= ! ConditionExpr
```

Example.

Notes.

6.8.6 Bitwise Complement Expression

```
BitwiseCompExpr ::= [ ] UnitExpr
```

Example.

Notes.

6.8.7 New Expression

```
NewExpr ::= new UnitExpr
```

Example.

Notes.

6.8.8 Dereference Expression

A *dereference expression* accepts an argument of reference type and returns a value (or element) of the reference's target type. The *dereference operator* returns the value referenced by the argument.

```
DereferenceExpr ::= * TermExpr
                  | TermExpr -> Identifier
```

Example. The following illustrates the dereference operator:

This function traverses a linked list counting the number of links it contains. The arrow operator is used to access the next link in the chain.

Notes.

6.9 Constructor Expressions

```
TermExpr ::= SetExpr
           | MapExpr
           | ListExpr
           | RecordExpr
```

6.9.1 Set Constructors

```
```

Example.

Notes.

6.9.2 Map Constructors

```
```

Example.

Notes.

6.9.3 List Constructors

```
```

Example.

Notes.

6.9.4 Record Constructors

```
```

Example.

Notes.

6.10 Term Expressions

A *terminal expression* is one which can terminate an expression tree (though does not necessarily do so). For example, a numeric literal represents a terminal node in an expression tree.

```
TermExpr ::= Ident
           | Literal
```

6.11 Lambda Expression

```

```

Example.

Notes.

Chapter 7

Flow Typing

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. Whiley's *type system* governs how the type of any variable or expression is determined. Whiley's type system is unusual in that it operates in a *flow-sensitive* manner allowing variables to have different types at different program points.

Chapter 8

Definite Assignment

See error reported for this check §10.3.4.

```
function f(int x) => int:
    int y
    if x < 0:
        y = 1
    return x + y
```

```
function f(int x) => int:
    int y
    while x < 0:
        y = 1
        x = x + 1
    return x + y
```

Chapter 9

Verification

The Whiley programming language supports *specifications* on functions, methods and data types which can be *statically verified* at compile time. Verification operates in an intra-procedural fashion based on a modified and extended version of Hoare logic[19]. To benefit from verification, programmers must provide specifications for their functions, methods and data types; additionally, they must provide loop invariants and other *assertion*s to guide the verifier.

Chapter 10

Error Messages

10.1 Parse Errors

10.2 Declarations

10.2.1 "Cyclic Constant Declaration" (301)

A *cyclic constant declaration* occurs when a constant declaration refers to itself, either *directly* or *indirectly*. This is an error because constants must be evaluated at *compile time*.

Example. The following illustrates several cyclic constant declarations:

```
constant const1 is 1 + const1

constant const2 is 1 + const3
constant const3 is 1 + const2
```

Here, all three constant declarations are cyclic. The declaration for `const1` has a *direct* cycle, because its definition refers to itself. The declaration for `const2` has an indirect cycle, because its definition refers to `const3` which, in turn, refers back to `const2`.

10.3 Expressions

10.3.1 "Invalid File Access" (000)

10.3.2 "Invalid Package Access" (000)

10.3.3 "Resolution Error" (000)

10.3.4 "Variable Possibly Uninitialised" (000)

A *variable possibly uninitialised* occurs when a variable may be used without being defined. That is, when a simple path exists through the control-flow graph of a function or method from that variable's declaration to a use which contains no definition for that variable. This error is reported as part of the *definite assignment* checking performed during compilation (see §8).

Example. The following illustrates a variable which is possibly uninitialised:

```
function f(int x) => int:
   int y
   return x + y
```

Here, variable `y` is definitely uninitialised in the expression "`x + y`". For more examples of variables which are possibly uninitialised, see §8.

10.3.5 "Ambiguous Coercion" (000)

10.4 Control Flow (000)

10.4.1 "Invalid LVal" (000)

10.4.2 "Invalid Tuple LVal" (000)

10.4.3 "Break Outside of Loop" (000)

10.4.4 "Unknown Variable" (000)

10.4.5 "Unknown Function or Method" (000)

10.4.6 "Variable Already Defined" (000)

A *variable redefinition* occurs when a variable is declared with a name matching another variable already in scope. This is an error because it is not permitted for one variable to shadow another.

Example. The following illustrates an example of a variable redefinition:

```
function add([int] xs, int v) => [int]:
    for k,v in xs:
        xs[k] = xs[k] + v
    return xs
```

Here, the **for** loop attempts to declare a variable v, but another variable v was already declared as a parameter.

10.4.7 "Duplicate Default Label" (000)

A *duplicate default label* occurs when a **switch** statement includes more than one **default** label. This is an error because at most one **default** is permitted.

Example. The following illustrates an example of a duplicate **default** label:

```
function f(int x):
    switch x:
        case 0:
            return 0
        default:
            return 1
        default:
            return 2
```

Here, the **switch** statement has two **default** labels. This must be an error as, otherwise, it would be ambiguous as to which executed.

10.4.8 "Duplicate Case Label" (000)

A *duplicate case label* occurs when a **switch** statement includes more than one **case** label matching the same value. This is an error because at most one **case** matching a given value is permitted.

Example. The following illustrates an example of a duplicate **case** label:

```
function f(int x):
    switch x:
        case 0:
            return 0
        case 0,1:
            return 1
        default:
            return 2
```

Here, the **switch** statement has two **case** labels, both of which match the value `0`. This must be an error as, otherwise, it would be ambiguous as to which executed.

10.4.9 "Unreachable Code" (000)

In a function or method, *unreachable code* arises when no possible execution path could reach them.

Example. The following illustrates some unreachable code:

```
function abs(int x):
    //
    if x < 0:
        return -x
    else:
        return x
    //
    return 0 // unreachable
```

Here, the final **return** statement can never be reached by any execution path through the `abs()` function. This is considered an error because it indicates something undesirable which may not have been intended.

10.4.10 “Missing Return Value” (000)

10.4.11 “Branch Always Taken” (000)

10.5 Functions (000)

10.5.1 “Reference Not Permitted in Function” (000)

10.5.2 “Method Invocation Not Permitted In Function” (000)

10.5.3 “Reference Access Not Permitted in Function” (000)

10.5.4 “Return from Void” (000)

10.6 Types (000)

10.6.1 “Subtype Error” (000)

10.6.2 “Incomparable Operands” (000)

10.6.3 “Record Type Required” (000)

10.6.4 “Record Missing Field” (000)

Glossary

assertion An assertion statement is specified with the **`assert`** keyword and identifies a condition which must hold at that point for all possible executions. 67

block comment A block comment begins with “/*” and continues until the end-of-comment marker “`*/`”. 10

boolean expression An expression which evaluates to a value of type **`bool`**. 17, 38, 39, 44, 76

compilation group A group of one or more source files being compiled together. 15, 75

compilation unit A single unit of compilation. In Whiley, this includes source files and also binary WyIL files. 15–17

compile time The point in time at which a given compilation group is compiled into binary form.. 69

compound statement A statement (e.g. **`if`**, **`while`**, etc) which may contain blocks of other statements. 37

constant declaration A source-level declaration which associates a name with a constant expression. The full name of the declared entity is determined from the package and name of the enclosing source file.. 16, 69

default package The top-level package which has no name, and is considered to be a “global” package.. 15

expression A combination of constants, variables and operators that, when evaluated, produce a single value. Expressions in certain circumstances may have side effects. 49, 75, 77

fault A fault is raised when an unrecoverable error in the program occurs. For a verified program, no faults are possible except to indicate an out-of-memory failure.. 38, 39, 56

function declaration A source-level declaration which defines a named function. The full name of the declared entity is determined from the package and name of the enclosing source file.. 16

indentation syntax A lexical organisation of source files where indentation is significant and is used to group statements and blocks. 9

intersection type A type formed by combining two or more types together (e.g. `[int]&[any` such that it includes any value contained in both. 21

line comment A line comment begins with "//" and continues until the end of line. 10

literal A source-level entity which describes a value of primitive type. 11

loop invariant A boolean expression which must hold on every iteration of a loop. 18, 20, 41, 42, 44, 67

method declaration A source-level declaration which defines a named method. The full name of the declared entity is determined from the package and name of the enclosing source file.. 16

name resolution The process of determining the fully qualified name of an identifier within a source file. Names are first resolved within the same source file, and then by searching the list of imported entities in reverse order. 16

negation type A type formed from another (e.g. `!int`), such that it includes any value not contained in the other. 21

package A unit of hierarchical organisation within the Whiley namespace.. 15

postcondition A logical condition over the parameters and returns of a function or method which must be true immediately after execution of that function or method.. 18–20, 32, 45

precondition A logical condition over the parameters of a function or method which must be true immediately prior to execution of that function or method.. 18–20, 32

safety critical system A system which operates in a high-risk setting where failure can lead to loss of life, injury, significant damage or environmental harm. 7

source file A file in which source code is located. Source files for the Whiley programming language have the extension `.whiley`. In Whiley, source files must be compiled into a binary form before they can be executed.. 9, 15, 17–19, 75–77

statement An program instruction which has an effect on the environment when executed, but does not produce a value. 37, 77

statement block A sequence of zero or more consecutive statements with the same indentation. 10, 37, 44

type An abstract entity which represents the set of values a given variable may hold, or a given expression may evaluate to.. 21, 76, 77

type declaration A source-level declaration which associates a name with a type descriptor. The full name of the declared entity is determined from the package and name of the enclosing source file.. 16

type descriptor A source-level description of an underlying type. Unlike many languages, type descriptors and types are quite distinct in Whiley as, for example, two distinct descriptors may describe the same underlying type. 21, 77

type pattern A source-level description of an underlying type (similar to a type descriptor) where one or more variables are associated with its subcomponent(s).. 22

union type A type formed by combining two or more types together (e.g. **`int|null`**), such that it includes any value contained in either. 21

value A value is an instance of a given type and permits a specific set of operations. Examples include: the integer value `1`; the list value `[1,2]`; and the **`null`** value.. 49

variable declaration A statement which declares one or more variable(s) for use in a given scope. Each variable is given a type which limits the possible values it may hold, and may not already be declared in an enclosing scope. 47, 77

variable initialiser An optional expression used to initialise variable(s) declared as part of a variable declaration. 47

verifying compiler A compilers which employs automated mathematical and logical reasoning to check the correctness of the programs that it compiles. 7

WyIL file A compiled (i.e. binary) form of a Whiley source file. 15, 75

Bibliography

[1] David J. Pearce. *Getting Started with Whiley*. 2014.

[2] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[3] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of November 1988. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 326–343, 1989.

[4] Software problem led to system failure at dhahran, saudi arabia, gao report #b-247094, 1992.

[5] Ariane 5: Flight 501 failure. report by the enquiry board. Technical report, European Space Agency, 1996.

[6] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[7] S. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[8] L. Peter Deutsch. *An interactive program verifier*. Ph.d., 1973.

[9] D. I. Good. Mechanical proofs about computer programs. In *Mathematical logic and programming languages*, pages 55–75, 1985.

[10] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford pascal verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

[11] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 1998.

[12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.

[13] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

[14] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The spec# programming system: An overview. Technical report, Microsoft Research, 2004.

[15] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the ACM conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41. ACM Press, 1993.

[16] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. volume 789 of *LNCS*, pages 687–706. 1994.

[17] Castagna and Frisch. A gentle introduction to semantic subtyping. In *Proc. ICALP*, pages 198–199, 2005.

[18] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *JACM*, 55(4):19:1–19:64, 2008.

[19] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12, 1969.