The Whiley Language Specification

Updated for version 0.3.38

David J. Pearce, 2014

Contents

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. For example, Whiley would be 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-–248, 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

- **The Whiley Rewrite Language (WyRL)**. David J. Pearce. In Proceedings of the Conference on Software Language Engineering (SLE), (to appear), 2015
- **Some Usability Hypotheses for Verification**. David J. Pearce. In Proceedings of the Workshop on Evaluation and Usability of Programming Languages (PLATEAU), (to appear), 2015
- **Integer Range Analysis for Whiley on Embedded Systems**. David J. Pearce. In Proceedings of the IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, (to appear), 2015.
- **Designing a Verifying Compiler: Lessons Learned from Developing Whiley**. David J. Pearce and Lindsay Groves. In Science of Computer Programming, (to appear), 2015

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, ^ 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 block*s (§5.1).

2.3 Comments

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

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

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

```
1 // 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 ::= _ | Letter

 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 | final | finite | for | function | if | import
          | in | int | is | method | native | new | no | null
          | package | private | protected | public | requires
          | return | skip | some | switch | throw | this
          | 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 (§4.3).

```
Literal ::= NullLiteral
          | BoolLiteral
          | BinaryLiteral
          | IntLiteral
          | HexLiteral
          | CharacterLiteral
          | StringLiteral
```

2.6.1 Null Literal

The **`null`** type (§4.3.1) has a single value expressed as the **`null`** literal.

```
NullLiteral ::= null
```

2.6.2 Boolean Literals

The **`bool`** type (§4.3.2) has two values expressed as the `true` and `false` literals.

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

2.6.3 Binary Literals

The **`byte`** type (§4.3.3) has 256 values which are expressed as sequences of binary digits, prefixed with "`0b`" (e.g. `0b0101`, `0b1111_0101`, etc).

```
BinaryLiteral ::= 0 b ( 0 | 1 | _ )+
```

Binary 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. `0b00101` becomes `0b00000101`).

2.6.4 Integer Literals

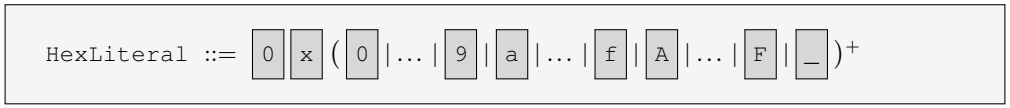
An *integer literal* is a sequence of numeric digits (e.g. `123456`, etc) corresponding to a value of **`int`** type (§4.3.4).

```
IntLiteral ::= ( 0 | ... | 9 | _ )+
```

Since integer values in Whiley are of arbitrary size (§4.3.4), there is no limit on the size of an integer literal.

2.6.5 Hexadecimal Literals

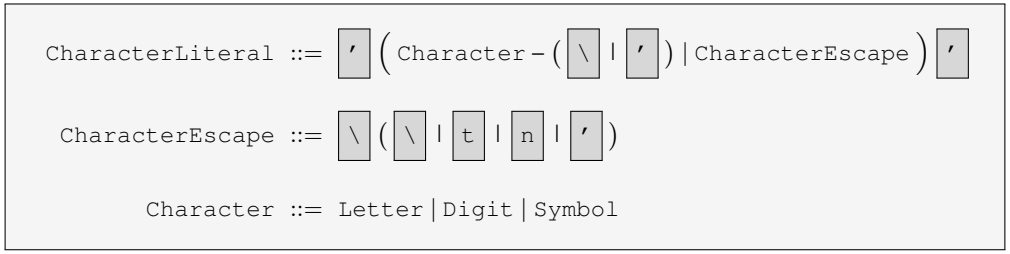
A *hexadecimal literal* is a sequence of hexadecimal digits (e.g. `0xffaf`, etc) corresponding to a value of **`int`** type (§4.3.4).



Since integer values in Whiley are of arbitrary size (§4.3.4), there is no limit on the size of a hexadecimal literal.

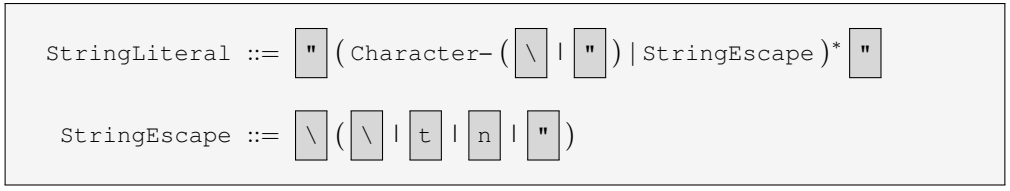
2.6.6 Character Literals

A *character literal* is expressed as a single character or an escape sequence enclosed in single quotes (e.g. `'c'`). Character literals generate integer constants corresponding to Unicode code points, which is necessary because there is no native character type.



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"`). String literals generate lists of integers corresponding to Unicode code points, which is necessary as there is no native string type.



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.

```
SourceFile ::= [PackageDecl]
               ( ImportDecl
               | (Modifier)* TypeDecl
               | (Modifier)* StaticVarDecl
               | (Modifier)* FunctionDecl
               | (Modifier)* MethodDecl
               )*
```

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 compiled 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.

```
PackageDecl ::= package Ident ( . Ident )*
```

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 declaration*s (§3.5.2), *constant declaration*s (§3.5.3), *function declaration*s (§3.5.4) and *method declaration*s (§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`

```
1 package g2d
2
3 type Point is { int x, int y }
4
5 constant Origin is { x: 0, y: 0 }
```

This declares two entities: `g2d.Graphics.Point` and `g2d.Graphics.Origin`. 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:

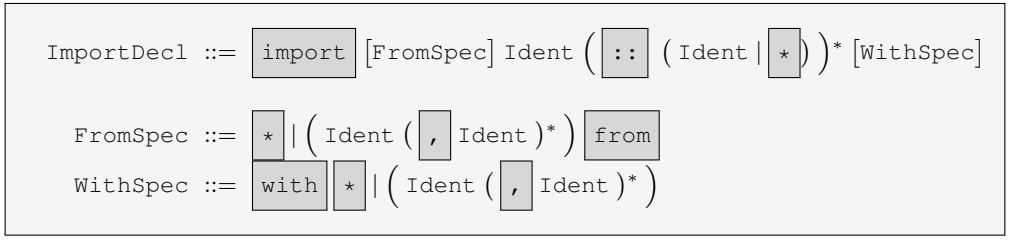
```
1 type Point is { int x, int y }
2
3 function Point(int x, int y) -> Point:
4     return {x: x, y: y}
```

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

Two named entities in the same category with different types are permitted in some circumstances, and this is referred to as *overloading*. Currently, overloading is only supported for entities representing function and methods or function and method types.

3.4 Imports

When performing *name resolution*, the 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:



A declaration of the form "`import a.pkg.File`" imports the compilation unit "`File`" in package "`a.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 a.pkg.File`" imports the named entity "`Entity`" from the compilation unit "`File`" residing in package "`a.pkg`". Note, this does *not* import the compilation unit "`a.pkg.File`" (and, hence, does not subsume the statement "`import a.pkg.File`"). In contrast, a declaration of the form "`import a.pkg.File with Entity`" imports both "`Entity`" and the compilation unit "`a.pkg.File`".

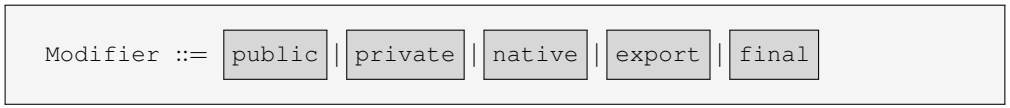
A *wildcard* may be used in place of the compilation unit name to import *all* compilation units within the given package (e.g. "`import some.pkg.*`"). A *wildcard* may be used in place of the entity name (e.g. "`import * from some.pkg.File`") to import *all* named entities within the given compilation unit.

3.5 Declarations

A *declaration* defines a new entity within a Whiley source file and provides a *name* by which it can be referred to within this source file, or from other source files.

3.5.1 Access Control

Several mechanisms for *access control* are provided through *declaration modifiers*.



- The **`public`** modifier declares that the declaration is visible from other Whiley source files.
- The **`private`** modifier declares that the declaration is visible only within the enclosing Whiley source file.
- The **`native`** modifier declares that the declaration is provided by the underlying system.
- The **`export`** modifier declares that the declaration is visible to source files written in other languages. Declarations with this modifier cannot be overloaded.
- The **`final`** modifier declares that the declaration cannot be reassigned.

When no modifier is given, the default of **`private`** is assumed.

Notes. The **`native`** and **`export`** modifiers together form the *foreign function interface*. The restriction on declarations declared with the **`export`** modifier is to enable names to be exported without *name mangling*.

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 [ Type | ( Variable ) ] ( where Expr )*

Variable ::= Type Ident
```

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* which ranges over the declared variable (if provided).

Examples. Some simple examples illustrating type declarations are:

```
1 // Define a simple point type
2 type Point is { int x, int y }
3
4 // Define the type of natural numbers
5 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. All types are also required to be *contractive*. This means, for example, that the declaration **`type`** `X` **`is`** `X` is considered invalid.

3.5.3 Static Variable Declarations

A *static variable declaration* declares a top-level variable within a Whiley source file with an optional initialiser expression.

```
StaticVarDecl ::= Type Ident [= Expr]
```

The given *initialiser expression* may not directly or indirectly refer to itself and may only be omitted in conjunction with the **`native`** modifier. Initialiser expressions are also *pure* and may not have *side-effects* (i.e. invoke methods or allocate on the heap).

Examples. Some simple examples to illustrate static variable declarations are:

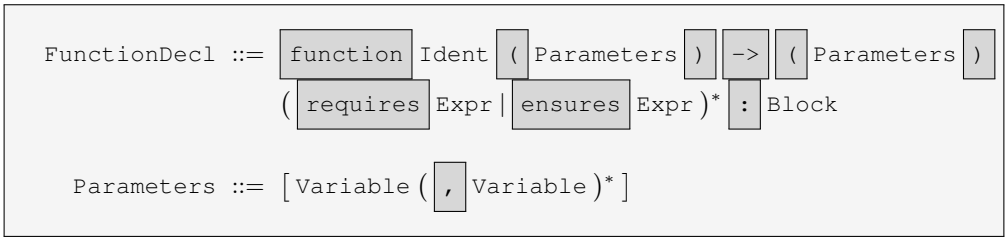
```
1 // Define a well-known mathematical constant!
2 final int TEN = 10
3
4 // Define an initialised global variable
5 int height = TEN * 2
```

The first declaration defines the constant `TEN` to have the **`int`** value `10`. The second declaration defines a global variable initialised with the constant.

Notes. Since initialiser expressions across a compilation group form a directed acyclic graph, static variables can always be safely initialised.

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 explicitly allocate memory on the heap and/or instigate concurrent computation.



Those variables declared before "`->`" are referred to as the *parameters*, whilst those declared afterwards are referred to as the *returns*. There are two 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 declared parameters. Multiple clauses may be given, and these are taken together as a conjunction. 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 declared parameters or returns. Multiple clauses may be given, and these are taken together as a conjunction. The convention is to specify **`ensures`** clause(s) after **`requires`** clause(s).

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

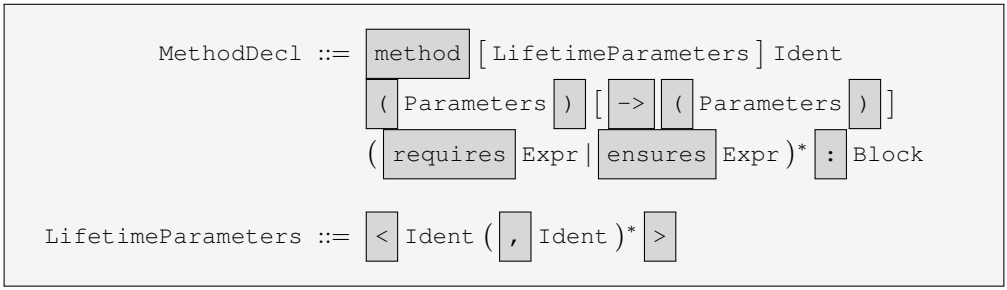
```
1  function max(int x, int y) -> (int z)
2  // return must be greater than either parameter
3  ensures x <= z && y <= z
4  // return must equal one of the parmaeters
5  ensures x == z || y == z:
6      // implementation
7      if x > y:
8          return x
9      else:
10         return y
```

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, methods may call other functions and/or methods (including **`native`** methods). They may also explicitly allocate memory on the heap, and/or instigate concurrent computation.



Those variables declared before "`->`" are referred to as the *parameters*, whilst those declared afterwards are referred to as the *returns*. The two optional clauses are defined identically as for function declarations (§3.5.4).

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

```
1  // Define the well-known concept of a linked list
2  type LinkedList is null | &{ LinkedList next, int data }
3
4  // Define a method which inserts a new item onto the end of the list
5  method insertAfter(LinkedList list, int item) -> LinkedList:
6      if list is null:
7          // reached the end of the list, so allocate new node
8          return new { next: (LinkedList) null, data: item }
9      else:
10         // continue traversing the list
11         list->next = insertAfter(list->next, item)
12         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 (§4.9), *intersection type*s (§**??**) and *negation type*s (§**??**), 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:

```
1 function id(null|int x) -> int|null:
2     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 semantic level. In this case, we have two distinct type descriptors which describe the *same* underlying semantic 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 ::= UnionType
            | TermType

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

4.3 Primitive Types

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

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

4.3.1 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. Values of **`null`** type support only equality comparators (§6.7). 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:

```
import std.math

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 (e.g. for terminating recursive types) and, in Whiley, is treated in a completely safe manner (unlike e.g. Java).

4.3.2 Booleans

The **bool** type represents the set of boolean values (i.e. `true` and `false`). Values of **bool** type support equality comparators (§6.7), binary logical operators (§6.10) and logical not (§6.10.1).

```
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 list[i] == item:
            return true
        i = i + 1
    // done
    return false
```

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

4.3.3 Bytes

The type **`byte`** represents the set of all 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. Values of **`byte`** type support equality comparators (§6.7), bitwise operators (§6.5), bitwise complement (§6.5.1) and shift operators (§6.5.3).

```
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 = ['0'; 8]
    int i = 0
    while i < 8:
        if (b & 0b00000001) == 0b00000001:
            r[i] = '1'
        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()`).

4.3.4 Integers

The type **int** represents the set of all arbitrary-sized integers, whose values are expressed as a sequence of one or more numerical or hexadecimal digits (e.g. `123456`, `0xffaf`, etc). Values of **int** type support equality comparators (§6.7), relational comparators (§6.3.2), additive (§6.3.3), multiplicative (§6.3.4) and negation (§6.3.1) operations.

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 function 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).

4.3.5 Void

The **void** type represents the empty set of values. Thus, **void** is the *bottom type* (i.e. $\bot$) 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 array. Finally, unlike the majority of other types, there are no *values* of type **void**.

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

```
// Attempt to update first element
method update1st(&(int[]) list, int value):
    // First, check whether list is empty or not
    if (*list) != [0;0]:
        // 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 `[0;0]` (i.e. the *empty array*).

4.4 Records

A record type describes the set of all compound values made from one or more *fields*, each of which has a unique name and a corresponding type. Values of record type support equality comparators (§6.7) and field access (§6.11.1) operations, as well as field assignment (§5.2.2).

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

MixedType ::= Type Ident
            | function Ident ParameterTypes -> ParameterTypes
            | method Ident ParameterTypes [ -> ParameterTypes ]
```

Records use *mixed types* for defining fields, 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:

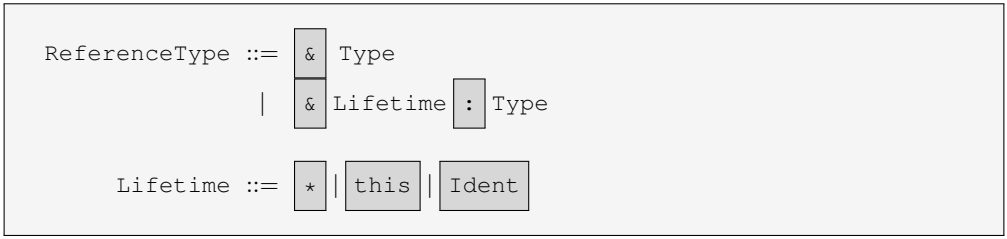
```
type Writer is {
    method write(byte[]) -> int,
    ...
}
type PrintWriter is {
```

```
    method write(byte[]) -> int,
    method println(ascii.string),
    ...
}
```

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`**").

4.5 References

A reference types represents the set of all references to values of a type given, 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 value of type `T`. Values of reference type support equality comparators (§6.7) and dereference (§6.12.2) operations, as well as dereference assignment (§5.2.2).



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.

4.6 Nominals

Nominal types represent user-defined types declared within one or more Whiley source files. Nominal types provide a mechanism for enforcing *information hiding*, and also for constructing *recursive types* (§4.10). All nominal types have an underlying — or, *concrete* — type and are indistinguishable from this type.

```
NominalType ::= Ident
```

Example. The following example illustrates nominal types:

```
1 // Using a nominal type to construct a recursive type
2 type LinkedList is null | { int data, LinkedList next }
```

The type `LinkedList` is declared using a reference to itself to define a recursive type (§4.10).

4.7 Arrays

An array type represents the set of all arrays holding values of a given element type. For example, `[1,2,3]` is an instance of array type `int[]`; however, `[1.345]` is not. Values of array type support equality comparators (§6.7) and access expressions (§6.4.2).

```
ArrayType ::= Type [ ]
```

Example. The following example illustrates array types:

```
1 function add(int[] v1, int[] v2) -> (int[] v3)
2 requires |v1| == |v2|
3 ensures |v1| == |v3|:
4     //
5     int i=0
6     while i < |v1|:
7         v1[i] = v1[i] + v2[i]
8         i = i + 1
9     return v1
```

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

4.8 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. Values of function or method type support equality comparators (§6.7) only.

```
FunctionType ::= function ParameterTypes -> ParameterTypes

MethodType ::= method [ ContextLifetimes ][ LifetimeParameters ]
               ParameterTypes [ -> ParameterTypes ]

ParameterTypes ::= ( [ Type ( , Type )* ] )
ContextLifetimes ::= [ ContextLifetime ( , ContextLifetime )* ]
ContextLifetime ::= this | Ident
```

Example. The following example illustrates function types:

```
1  type Fun is function(int) -> int
2
3  function map(int[] items, Fun fn) -> int[]:
4      //
5      int i = 0
6      while i < |items|:
7          items[i] = fn(items[i])
8          i = i + 1
9      //
10     return items
```

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

4.9 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 comparators (§6.7) and type tests (§6.14). See §4.11 for exceptions to this.

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

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`).

4.10 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 (§4.6).

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.

4.11 Effective Types

An effective type is a union of types which all contain some property (e.g. a union of arrays). This common property allows the effective type to support more operations than possible for an arbitrary union (§4.9).

4.11.1 Effective Records

An effective record is a union of two or more record types with at least one field in common. For example, `{int f, int g}|{real f, int h}` 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.

4.11.2 Effective Array

An effective array is a union of array types. For example, `int[]|real[]` is an effective array. An effective collection supports all operations valid for a array type (§4.7). 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.

4.12 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 § 4.12.2 for more).

4.12.1 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:

- `{int|null f}` is equivalent to `{int f}|{null f}`

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

4.12.2 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 `int|bool`. 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 *anti-symmetric* 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 *statement*s, which cause effects on the environment. However, statements in Whiley do not produce values. *Compound 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 (§2.2). Statement blocks are used to group statements together when constructing compound statements. For example:

```
1  function sum(int[] items) -> int:
2      // outer block begins
3      int r = 0
4      int i = 0
5      while i < |items|:
6          // inner block begins
7          r = r + items[i]
8          i = i + 1
9          // inner block ends
10     //
11     return r
12     // 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.

5.1.1 Named Blocks

A *named block* identifies a distinct subregion of a given block. The primary use for named blocks is in the context of reference lifetimes (§4.5), as these can be associated with an enclosing block.

```
NamedBlock^ℓ ::= Ident : Block^γ

(where ℓ < γ)
```

Example. The following illustrates a named block:

```
1 method m(int item):
2   &this:int x = this:new item
3   myblock: // declares a new lifetime 'myblock'
4     &myblock:int y = x
```

Here, a named block is used to constrain the lifetime of reference `y`. As a result, the lifetime of reference `x` strictly contains that of reference `y`.

5.2 Simple Statements

A *simple statement* is a statement where control always continues to the next statement in sequence. Simple statements do not contain other statements nested within them.

5.2.1 Assert Statement

An *assert statement* is 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:

```
1 function abs(int x) -> int:
2     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.

5.2.2 Assignment Statement

An *assignment statement* is of the form `leftHandSide = rightHandSide`. Here, the `rightHandSide` is a sequence of one or more expressions, whilst the `leftHandSide` is sequence to an identical number of `LVal` expressions — that is, expressions permitted on the left-hand side of an assignment. At runtime, those values generated by evaluating the right-hand side must be subtypes (§4.12.2) of their corresponding target on the left-hand side. An assignment statement which contains multiple `LVal` expressions on the left-hand side is referred to as a *multiple assignment*. An assignment statement which contains an `LVal` expression on the left-hand side consisting purely of an `Ident` is said to *directly assign* that variable.

```
AssignStmt ::= LVal ( , LVal )* = Expr ( , Expr )*

      LVal ::= Ident
             | LVal . Ident
             | LVal [ Expr ]
             | * Expr
```

Example. The following illustrates different possible assignment statements:

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

method f2({int f} x, int y):
    x.f = y        // field assignment

method f3(int[] x, int i, int y):
    x[i] = y       // list assignment

method f4({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 arrays and records.

5.2.3 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:

```
1 function abs(int x) -> (int y) ensures y >= 0:
2     //
3     assume x >= 0
4     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.

5.2.4 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 (§5.2.5).

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

Example. The following illustrates a `debug` statement:

```
1 function f(int x) -> int:
2     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 report when a given function is invoked.

5.2.5 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.

5.2.6 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 ::= Type Ident ( , Type Ident )* [ = Expr ( , 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

5.3 Control Statements

A *control statement* is a statement which may have multiple exit points, and where control does not always continue to the next statement in sequence. Control statements may contain other statements nested within them.

5.3.1 Break Statement

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

```
BreakStmt ::= break
```

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

```
// Find first index matching x
function find(int[] xs, int x) -> int:
    int i = 0
    while i < |xs|:
        if xs[i] == x:
            break
        else:
            i = i + 1
    //
    return i
```

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 (§5.3.7). This is because **switch** statements have *explicit*, rather than *implicit*, fall-through.

5.3.2 Continue Statement

A *continue statement* can be used either to transfer control to the next iteration of the enclosing loop (i.e. **`do`**, **`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 that 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 (§5.3.7). This is because **`switch`** statements have *explicit*, rather than *implicit*, fall-through.

5.3.3 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^ℓ ::= do : Block^γ while Expr ( where Expr )*

                                                (where ℓ < γ)
```

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 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`** (§5.3.1) statement.

5.3.4 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.

5.3.5 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*.

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

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

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

```
1 function max(int x, int y) -> int:
2     if(x > y):
3         return x
4     else if(x == y):
5         return 0
6     else:
7         return y
```

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

5.3.6 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.

$$\texttt{ReturnStmt} ::= \boxed{\texttt{return}}\ \big[\ \texttt{Expr}\ \big(\ \boxed{\texttt{,}}\ \texttt{Expr}\ \big)^{*}\ \big]$$

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.

5.3.7 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 expression. 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.

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

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

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

(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`).

5.3.8 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.

$$\texttt{WhileStmt}^{\ell} ::= \boxed{\texttt{while}}\ \texttt{Expr}\ \big(\ \boxed{\texttt{where}}\ \texttt{Expr}\ \big)^{*}\ \boxed{\texttt{:}}\ \texttt{Block}^{\gamma} \qquad (\text{where } \ell < \gamma)$$

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** (§5.3.1) statement.

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

The operands for operators in Whiley are evaluated in a specific left-to-right *evaluation order*. This always respects parentheses and operator precedence. Furthermore, aside from the short-circuiting operators (§6.10.2), operands are always fully evaluated before any part of the operation is performed.

6.1.1 Operator Precedence

To determine the evaluation order for mixed-operator expressions without explicit parenthesis, a fixed *operator precedence* is used. This is first determined by *operator class*:

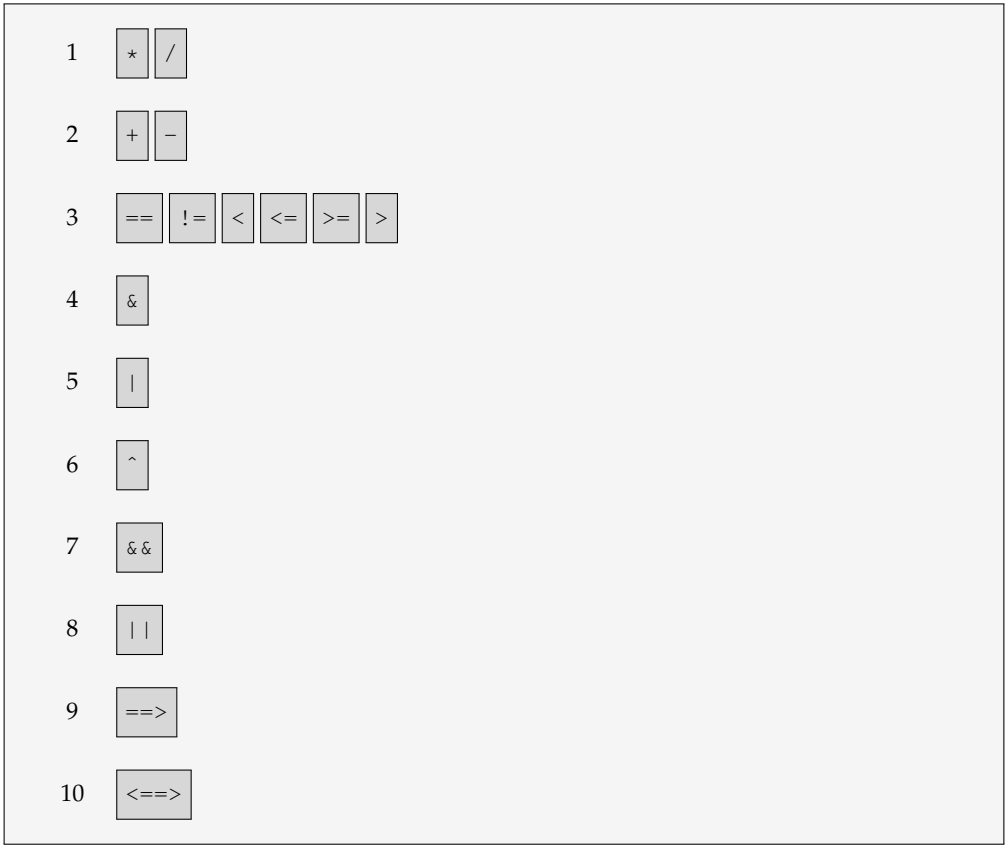
1. **Unary Expressions.** This operator class represents operators which take exactly one operand. This class takes highest precedence, and includes operators such as arithmetic negation (§6.3.1) and logical not (§6.10.1).

2. **Binary (Infix) Expressions.** This operator class represents operators which accept two operands with an infix syntax. This class includes the usual range of common binary operators, such as arithmetic operators (§6.3.3,§6.3.4), logical connectives (§6.10.2), etc.

3. **Binary (Mixfix) Expressions.** This operator class represents operators which accept two operands but which are non-infix operators and, hence, precedence is not ambiguous. This class includes the array access (§6.4.2) and field access operator (§6.11.1).

4. **N-Ary Expressions.** This operator class represents operators which accept an arbitrary number of operands. This class includes array constructors (§6.4.4), record constructors (§6.11.2), etc.

Within the class of binary infix expressions, an explicit precedence rank is given for each operator:



Lower ranked operators bind more tightly (i.e. take higher precedence) than higher ranked operators.

6.2 Unit Expressions

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

```
Expr ::= ArithmeticExpr
       | BitwiseExpr
       | CastExpr
       | EqualityExpr
       | InvokeExpr
       | LambdaExpr
       | LogicalExpr
       | ArrayExpr
       | RecordExpr
       | ReferenceExpr
       | TermExpr
```

6.3 Arithmetic Expressions

Arithmetic expressions operate on values of numeric type (currently just **`int`**).

```
ArithmeticExpr ::= ArithmeticNegationExpr
                 | ArithmeticRelationalExpr
                 | ArithmeticAdditiveExpr
                 | ArithmeticMultiplicativeExpr
```

6.3.1 Negation Expressions

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.

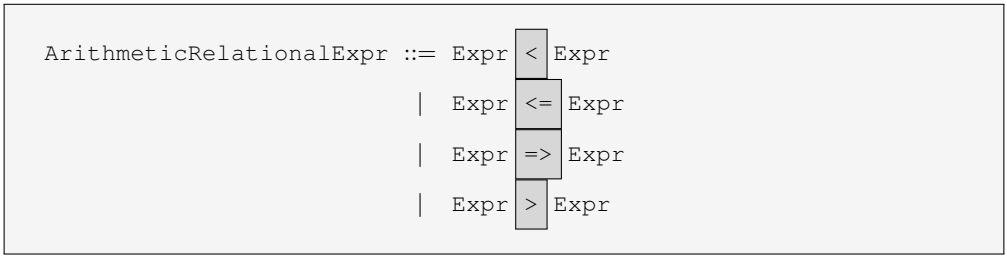
```
ArithmeticNegationExpr ::= - Expr
```

Example. The following illustrates the negation operator:

```
1 function negAccess(int i, int[] items) -> int
2 requires -|items| <= i && i < |items|:
3    //
4    if i < 0:
5        return -items[-(i+1)]
6    else:
7        return items[i]
```

6.3.2 Relational Expressions

Relational 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.



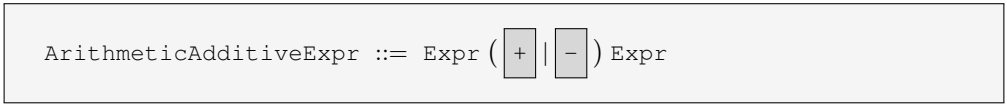
Example. The following example illustrates the strict inequality comparators:

```
function compare(int x, int y) -> int:
    if x < y:
        return -1
    else if x > y:
        return 1
    else:
        return 0
```

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.

6.3.3 Additive Expressions

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



Example. The following illustrates the additive operators:

```
function diff(int a, int b) -> int:
    return a - b
```

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

6.3.4 Multiplicative Expressions

A multiplicative expression accepts two arguments of type **`int`** and produces a result of the same 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.

```
ArithmeticMultiplicativeExpr ::= Expr ( * | / | % ) Expr
```

Example. The following illustrates the remainder operator:

```
1 function indexOf(int[] xs, int i) -> int
2 requires i >= 0 && |xs| > 0:
3     //
4     return xs[i % |xs|]
```

This function accepts a non-negative integer and uses this to index into an array. To ensure the array access is within bounds, the remainder operator is used. Furthermore, the function requires the array 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`).

6.4 Array Expressions

Array expressions operate on values of array type (e.g. **`int`**`[]`, `(`**`bool`**`|`**`byte`**`)[]`, etc).

```
ArrayExpr ::=
            | ArrayLengthExpr
            | ArrayAccessExpr
            | ArrayGeneratorExpr
            | ArrayInitialiserExpr
```

6.4.1 Length Expressions

The *lengthof operator* accepts a value of array type, and produces a value of **int** type which equals the number of elements in the array.

```
ArrayLengthExpr ::= | Expr |
```

Example. The following example illustrates the lengthof operator:

```
// Return first item in list over a given item
function firstOver(int[] items, int item) -> int|null:
    int i = 0
    while i < |items|:
        if items[i] > item:
            return item
        i = i + 1
    // no match
    return null
```

The above function iterates through all elements in an array looking for the first which is above a given item. The length operator is used to ensure this iteration remains within bounds.

6.4.2 Access Expressions

An array access expression accepts a array argument with one operand and produces a value of the array element type. The *index-of operator* returns the element at the given operand position in the array.

```
ArrayAccessExpr ::= Expr [ Expr ]
```

Examples. The following example illustrates the array access operator:

```
// Check whether an array is sorted or not
function isSorted(int[] items) -> bool:
    int i = 1
    //
    while i < |items|:
        if items[i-1] > items[i]:
            return false
        i = i + 1
```

```
    //
    return true
```

The above function determines whether a given array of integers is sorted from smallest to largest. The array access operator is used to access successive elements in the array.

6.4.3 Generator Expressions

An *array generator* accepts two arguments and produces a value of array type. The second argument must be of **int** type and the array produced contains exactly this many occurrences of the first argument.

```
ArrayGeneratorExpr ::= [ Expr ; Expr ]
```

Examples. The following example illustrates an array generator:

```
function cons(int head, int[] tail) -> int[]:
    int[] r = [head; |tail| + 1]
    int i = 0
    //
    while i < |tail|:
        r[i+1] = tail[i]
        i = i + 1
    //
    return r
```

This function constructs a array by prepending a given element onto the front of a given array. The array generator is used to construct the initial array of values whose size is one larger than the original array.

6.4.4 Array Initialiser

An *array initialiser* accepts zero or more operands and produces a value of array type. Array initialisers are used to construct arrays from their constituent elements.

```
ArrayInitialiserExpr ::= [ [ Expr ( , Expr )* ] ]
```

Example. The following example illustrates an array initialiser:

```
constant digits is ['0','1','2','3','4','5','6','7','8','9']

// Convert an integer value into a string
function toString(int item) -> int[]:
    int[] r = [0;0]
    //
    while item != 0:
        int v = item / 10
        int w = item % 10
        r = Arrays.append(digits[w],r)
        item = v
    //
    return r
```

The above function converts an integer value into its string representation. An array initialiser is used to map integer values to their corresponding digits. An empty array initialiser is also used to initialise the string.

6.5 Bitwise Expressions

Bitwise expressions operate on values of **byte** type.

```
BitwiseExpr ::= BitwiseComplementExpr
              | BitwiseBinaryExpr
              | BitwiseShiftExpr
```

6.5.1 Complement Expressions

The *bitwise complement operator* accepts an argument of **byte** type (§4.3.3) and produces a result of matching type. The operator returns bitwise complement of the argument; that is, where the sign of each bit is reversed.

```
BitwiseComplementExpr ::= ~ Expr
```

Example. The following example illustrates the bitwise complement operator:

```
// Check whether a given bit is zero
function isZero(byte b, int bit) -> bool:
    byte mask = 0b1 << bit
```

```
4       return (b & ~mask) == b
```

6.5.2 Binary Expressions

A bitwise binary expression operates on values of **byte** type (§4.3.3). The *bitwise AND* operator, `&`, performs a logical AND between the respective bits of each operand, and produces a **byte**. The *bitwise OR* operator, `|`, performs a logical OR 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**.

```
BitwiseBinaryExpr ::= Expr ( & | | | ^ ) Expr
```

Example. The following example illustrates the bitwise OR operator:

```
1  constant AF is 4
2  constant ZF is 6
3
4  function setFlag(byte flags, int flag) -> byte:
5      byte mask = 0b0000_0001 << flag
6      return flags | mask
7
8  function getFlag(byte flags, int flag) -> bool:
9      byte mask = 0b0000_0001 << flag
10     return (flags & mask) != 0
```

These functions provide mechanisms for manipulating a byte of "flags", as determined by the constant identifiers. The bitwise OR operator is used to ensure a given bit is set, whilst the bitwise AND operator is used to check whether one is set or not. This example also illustrates the left-shift operator (§6.5.3).

6.5.3 Shift Expressions

A bitwise 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`.

```
BitwiseShiftExpr ::= Expr [ ( « | » ) Expr ]
```

Examples. The following illustrates the left shift operator:

```
1  public function toUnsignedByte(u8 v) -> byte:
2      //
3      byte mask = 0b00000001
4      byte r = 0b0
5      int i = 0
6      while i < 8:
7          if (v % 2) == 1:
8              r = r | mask
9          v = v / 2
10         mask = mask << 1
11         i = i + 1
12     return r
```

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 Cast Expressions

A *cast operator* accepts a value of one type and returns a value of a different, but equivalent, type and this may result in a change of the underlying representation.

```
CastExpr ::= ( DefiniteType ) Expr
```

Example. The following illustrates a cast operator being used:

```
1
2  function f(Point3D p) -> Point2D:
3      return (Point2D) p
```

This function converts a record containing three *int* fields into one containing two *int* fields. This requires that each field in the latter is a valid field in the former.

6.7 Equality Expressions

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

```
EqualityExpr ::=
               | Expr == Expr
               | Expr != Expr
```

Example. The following example illustrates an equality expression:

```
function contains(int[] items, int item) -> bool:
    //
    int i = 0
    while i < |items|:
        if i == item:
            return true
        i = i + 1
    return false
```

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

6.8 Invoke Expressions

A *function or method invocation* executes a named function or method declared in a given source file. An *indirect function or method invocation* executes a function determined by a given expression. An invocation passes arguments of appropriate number and type to the executed function or method. An invocation may also return one or more values which can be subsequently used.

```
         InvokeExpr ::= Name [ LifetimeArgsList ] ArgsList
  IndirectInvokeExpr ::= Expr [ LifetimeArgsList ] ArgsList

            ArgsList ::= ( [ Expr ( , Expr )* ] )
    LifetimeArgsList ::= < Lifetime ( , Lifetime )* >
```

Example. The following example illustrates a function invocation:

```
// Determine the max of two values
function max(int x, int y) -> int:
    if x >= y:
        return x
    else:
        return y

// Determine the max of 1 or more values
function max(int[] items) -> int
requires |items| > 0:
    //
    int r = 0
    int i = 0
    while i < |items|:
        r = max(r,items[i])
        i = i + 1
    //
    return r
```

This example illustrates one function being called from another. Both functions have the same name and are said to *overload* one another. Function resolution identifies the appropriate function based on the number and type of arguments supplied.

6.9 Lambda Expressions

A *lambda expression* creates an anonymous function or method which can accept zero or more arguments and whose return type is inferred from the body of the lambda.

```
LambdaExpr ::= & [ ContextLifetimes ] [ LifetimeParameters ]
               ( [ Type Ident ( , Type Ident )* ] -> Expr )
```

Example. The following example illustrates a lambda expression:

```
// Type of function which accepts and returns an int
type fun_t is function(int)->int

// Apply a function to every element of a list
function map(fun_t fn, int[] xs) -> int[]:
    int i = 0
    while i < |xs|:
```

```
        xs[i] = fn(xs[i])
        i = i + 1
    return xs

// Add y to every element of items
function addAll(int[] items, int y) -> int[]:
    fun_t fn = &(int x -> x + y)
    return map(fn,items)
```

This function illustrates the classical *map* function which applies a function to all elements of a collection. In this case, a lambda is used to create a function which adds a constant value to its argument. This lambda is used to implement `addAll()` in terms of `map()`.

6.10 Logical Expressions

Logical expressions operate on values of **`bool`** type.

```
LogicalExpr ::=
              | LogicalNotExpr
              | LogicalBinaryExpr
              | LogicalQuantExpr
```

6.10.1 Not Expressions

The *logical not* operator accepts an argument of **`bool`** type and produces a value of **`bool`**. The value returned is the logical opposite of the argument.

```
LogicalNotExpr ::= ! Expr
```

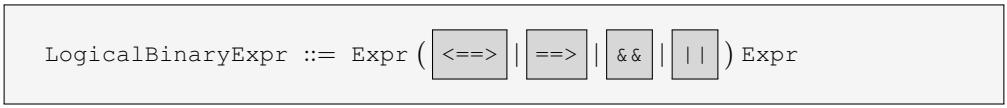
Example. The following example illustrates the logical not operator:

```
function max(int a, int b):
    if !(a < b):
        return a
    else:
        return b
```

This function computes the maximum of two **`int`** values. The expression `!(a < b)` is equivalent to `a >= b` and is used purely to illustrate the logical not operator.

6.10.2 Connective Expressions

A logical connective operates on values of **`bool`** type (§4.3.2) 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`.



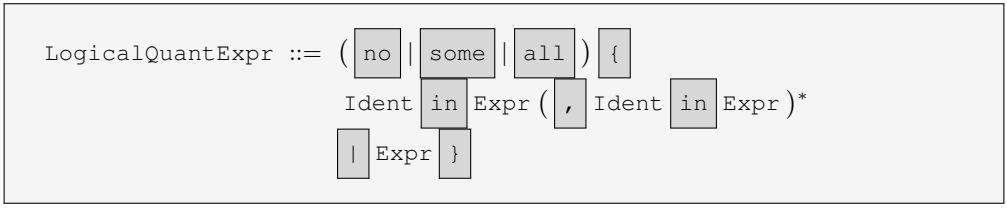
Example. The following examples illustrate some of the logical operators:

```
1 function implies(bool x, bool y) -> bool:
2     return !x || y
3
4 function iff(bool x, bool y) -> bool:
5     return implies(x,y) && implies(y,x)
```

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.

6.10.3 Quantifier Expressions

A quantifier operates over an array of values and produces a value of **`bool`** type. The *universal quantifier*, **`all`**, returns `true` if the given expression evaluates to `true` for every element in the array, and `false` otherwise. The *existential quantifier*, `some`, returns `false` if the given expression evaluates to `false` for every element in the array, and `true` otherwise. The *inverted universal quantifier*, **`no`**, returns `true` if the given expression evaluates to `false` for every element in the array, and `false` otherwise



Examples. The following example illustrates the universal quantifier:

```
1 // A type representing lists of natural numbers
```

```
type natlist is (int[] xs) where
    all { i in 0 .. |xs| | xs[i] >= 0 }
```

Here, the type `natlist` represents those integer arrays for which every element is a natural number (i.e. greater-or-equal to zero).

6.11 Record Expressions

Record expressions operate on values of record type (e.g. `{int x, int y}`, etc).

6.11.1 Access Expressions

The field access operator accepts a value of record type and returns the value held in a given field.

```
FieldAccessExpr ::= Expr [.] Ident
```

Examples. The following example illustrates a field access expression constructor:

```
type Vec is {int x, int y, int z}

function dotProduct(Vec v1, Vec v2) -> Vec:
    return (v1.x * v2.x) + (v1.y * v2.y) + (v1.z * v2.z)
```

The above function computes the so-called *dot product* of two vectors. The field access operator is used to access the three fields of each vector.

6.11.2 Record Initialisers

A *record initialiser* accepts one or more operands and produces a value of record type. Record constructors are used to construct records from their constituent elements.

```
RecordInitialiserExpr ::= [{] FieldArgsList [}]

        FieldArgsList ::= Ident [:] Expr ( [,] Ident [:] Expr )*
```

Example. The following example illustrates a record initialiser:

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

// Translate a given point based on a delta in x and y
function move(Point p, int dx, int dy) -> Point:
    return { x: p.x+dx, y: p.y+dy }
```

The above function simply translates a `Point` from one position to another based on a shift in `x` and in `y`. The record initialiser is used to construct the new `Point`.

6.12 Reference Expressions

Reference expressions operate on values of reference type (e.g. `&int`).

6.12.1 New Expressions

A new expression accepts an argument of any type and produces a reference to that type. The *new operator* allocates sufficient space on the heap and initialises it with the given value. It then returns a reference to this heap object.

```
NewExpr ::= new Expr
          | Lifetime : new Expr
```

Example. The following example illustrates the new operator:

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

// Add a new item onto the head of the list
method add(LinkedList list, int item) -> LinkedList:
    //
    return new {next: list, data: item}
```

This example illustrates an operation for adding an item onto the front of a classical linked list. Here, a `LinkedList` is either `null` or a reference to a node containing a `next` reference and `data` item. The add operation simply allocates a new node and places it on the front of the list.

6.12.2 Dereference Expressions

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

```
DereferenceExpr ::= * TermExpr
                  | Expr -> Ident
```

Example. The following illustrates the dereference operator:

```
1 type LinkedList is null | &{LinkedList next, int data}
2
3 method length(LinkedList l) -> int:
4     //
5     if l is null:
6         return 0
7     else:
8         return 1 + length(l->next)
```

This method 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. The arrow operation "`e->f`" is a short-hand notation for "`(*e).f`" and can be used when `*e` has effective record type (§4.11.1).

6.13 Terminal 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
           | ( Expr )
```

6.14 Type Test Expressions

```
TypeTestExpr ::= Expr is Type
```

Chapter 7

Type Checking

The Whiley programming language is *statically typed*, meaning that: firstly, every expression has a type determined at compile time; second, 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.

7.1 Overview

A *type environment*, Γ, binds variables declared in the enclosing scope(s) to their *current type*. The current type of a variable may be its declared type, or a refinement thereof. The environment $\Gamma[\mathtt{x} \mapsto \mathtt{T}]$ contains all of the bindings in Γ, except where `x` now binds to `T`. The initial type environment, Γ_0, for the **`requires`**, **`where`** clause(s) and body of a **`function`**, **`method`** or **`type`** declaration contains exactly one binding for each parameter to its declared type. The initial type environment, Γ_r, for the **`ensures`** clause(s) of a **`function`** or **`method`** additionally contains exactly one binding for each return to its declared type. For example, consider the following partial declaration:

```
1 function f(int x, bool y) -> (null|int r):
2     ...
```

Here, $\Gamma_0 = \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{bool}\}$ and $\Gamma_r = \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{bool}, \mathtt{r} \mapsto (\mathtt{int} \vee \mathtt{null})\}$.

7.1.1 Flow Typing

Whiley's type system employs flow-sensitive typing — *flow typing* — for determining the type of each local variable within a given statement block. The *pre-environment* gives the type environment immediately before a given statement. Likewise, the *post-environment* gives the type environment immediately after a given statement. The flow

typing system is responsible for calculating, for each statement, the post-environment from the pre-environment. The judgment $\Gamma_0 \vdash \mathtt{S} \dashv \Gamma_1$ indicates that typing statement $\mathtt{S}$ with environment Γ_0 produces the (potentially updated) environment Γ_1. For a given statement block $\mathtt{S}_1 \ldots \mathtt{S}_n$, it follows that $\Gamma_0 \vdash \mathtt{S}_1 \ldots \mathtt{S}_n \dashv \Gamma_n$ expands by chaining the post-environment for each statement $\mathtt{S}_n$ into its successor $\mathtt{S}_{n+1}$. That is, $\Gamma_0 \vdash \mathtt{S}_1 \dashv \Gamma_1$, $\Gamma_1 \vdash \mathtt{S}_2 \dashv \Gamma_2$, and so on.

7.1.2 Scoping

The *lifetime* of a local variable extends from its declaration within a given statement block to the end of that block. For example, if statement $\mathtt{S}$ declared variable $\mathtt{x}$ to have type $\mathtt{T}$, it follows that $\Gamma \vdash \mathtt{S} \dashv \Gamma[\mathtt{x} \mapsto \mathtt{T}]$. Furthermore, we require that $\mathtt{x}$ was not already declared in Γ (i.e. that $\mathtt{x} \notin \Gamma$). Observe that variables of the same name may be declared in different blocks, provided one is not nested within the other.

7.1.3 Environment Joining

At meet points in the control-flow graph of a statement block the typing environments from each branch must be *joined* together. If Γ_a and Γ_b are type environments then their join, denoted $\Gamma_a \sqcup \Gamma_b$, is a single environment carefully constructed from them. This join operator is defined as follows:

$$\Gamma_a \sqcup \Gamma_b \quad = \quad \{\mathtt{x} \mapsto (\mathtt{T}_a \vee \mathtt{T}_b) \mid \mathtt{x} \mapsto \mathtt{T}_a \in \Gamma_a \ \wedge \ \mathtt{x} \mapsto \mathtt{T}_b \in \Gamma_b\}$$

Every variable defined in both environments is bound in their join to the union of its type in each environment. The following illustrates a situation where joining is necessary:

```
1  function f(int|null x) -> (int r):
2     //
3     if x is null:
4        return 0
5     else:
6        x = x + 1
7     //
8     return x
```

The pre-environment for the **`return`** statement is formed from the post-environments of the true- and false-branches of the conditional. The former is $\{\mathtt{x} \mapsto \mathtt{void}\}$ and the latter is $\{\mathtt{x} \mapsto \mathtt{int}\}$ and the resulting join is $\{\mathtt{x} \mapsto \mathtt{int}\}$.

7.2 Type Refinement

In certain circumstances a runtime type test may result in *type refinement*. That is, where the type of a variable is refined from its current type (e.g. $\mathtt{T}_1$) to a more precise

type (e.g. $\texttt{T}_2$ where $\texttt{T}_2 \leq \texttt{T}_1$). More specifically when a type test "`e` **`is`** $\texttt{T}_2$" holds, the type of `e` may be refined to $\texttt{T}_1 \wedge \texttt{T}_2$. Likewise, when "`e` **`is`** $\texttt{T}_2$" does not hold, the type of `e` may be refined to $\texttt{T}_1 \wedge \neg\texttt{T}_2$. Type refinement may only occur when a type test is used as the conditional expression for an **`if`**, **`while`**, **`do`**–**`while`**, **`assert`** or **`assume`** statement. Furthermore, an expression `e` will be refined only if it is a *refinable expression*. A refinable expression is a variable access or a field access acting on a refinable expression. The following illustrates a common scenario:

```
1 function f(int|null x) -> (int r):
2    //
3    if x is int:
4       return x
5    else:
6       return 0
```

The initial environment for the body of `f()` is given by $\Gamma_0 = \{\texttt{x} \mapsto (\texttt{int} \vee \texttt{null})\}$. In this case, the type of variable `x` is refined to $(\texttt{int} \vee \texttt{null}) \wedge \texttt{int}$ in the true branch (which is equivalent to `int`) and $(\texttt{int} \vee \texttt{null}) \wedge \neg\texttt{int}$ in the false branch (which is equivalent to `null`) .

7.2.1 Expressions

Type refinement may occur within expressions when a given type test is known to hold or not. The following illustrates:

```
1 if x is int && x >= 0:
2    //
3 else:
4    //
```

Here, the type of variable `x` is refined to **`int`** within `x >= 0`. Observe, however, that no refinement occurs on the **`else`** branch as the given expression does not capture all possible integer values.

Since the logical connectives have short-circuiting behaviour, so does type refinement within expressions. That is, the refinement must occur before the expression where the refined type is required.

7.3 Function and Method Resolution

Look at the rules for determining which function or method is being selected.

- Most precise type selected
- If no unique precise type, then ambiguous

7.4 Coercions

Look at the rules for when a coercion is permitted or not.

Chapter 8

Definite (Un)Assignment

The Whiley programming language requires that variables are known at *compile time* to be *definitely assigned* (i.e. that they are defined before used) and, similarly, that **`final`** variables are *definitely unassigned* at the point of assignment (i.e. that they are assigned at most once). A conservative approach is taken to determining whether or not these requirements hold true. This ensures the language can be compiled efficiently, but also means that some provably safe programs are not valid Whiley programs. In this chapter, we specify the process by which definite assignment and unassignment are determined. The mechanism underpinning this is a data-flow analysis over the control-flow graph of a function to determine the assignment status of all local variables.

8.1 Overview

The following illustrates a simple function which will be rejected by the compiler because it cannot determine definite assignment for all variables. The function is said to *fail definite assignment*:

```
1 function f(int x) -> (int r):
2     int y
3     //
4     if x < 0:
5         y = 1
6     //
7     return x + y
```

In the above program, variable `y` is *not* definitely assigned before its use in the **`return`** statement. This is because there is an *execution path* through the function which reaches the **`return`** statement and on which variable `y` is not defined (see Figure 8.1). In fact, there are *two* possible execution paths through this function, but variable `y`

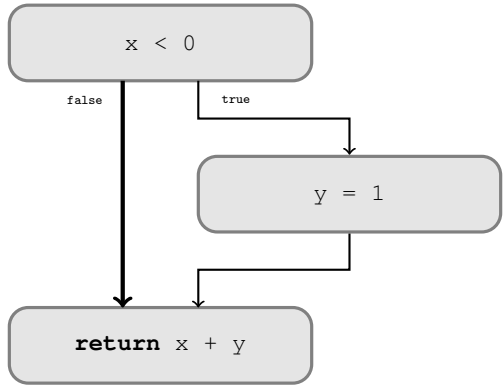


Figure 8.1: Control-Flow Graph for function `f()`. On the bold path, `y` is undefined.

is only defined on one of them. Observe that, since it is a parameter, variable `x` is automatically considered to have been defined on entry to the function.

In contrast, the following illustrates a simple function which will be rejected by the compiler because it cannot determine a **`final`** variable is definitely unassigned at the point of assignment:

```
1 function g(int x) -> (int r):
2     //
3     final int y = 0
4     //
5     if x < 0:
6         y = 1
7     //
8     return y
```

In the above program, the **`final`** variable `y` is *not* definitely unassigned at the assignment on line 5. In other words, there is an execution path through the function on which variable `y` is assigned more than once. Such a path violates the intention of the **`final`** modifier, which dictates that a variable should be assigned at most once.

8.1.1 Loops

The treatment of loops with respect to definite (un)assignment warrants special attention. Recall the mechanism for determining definite (un)assignment is conservative. In the context of loops, it simply assumes *every loop can be executed zero or more times* (i.e. even if this is not correct). From the perspective of definite assignment, this implies that a variable assignment within a loop *may not occur*. From the perspective of definite unassignment, this implies that a variable assignment within a loop *may occur*. The following illustrates:

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

The above function fails definite assignment because variable `y` is not defined when zero iterations of the loop are executed (e.g. when `x==0` on entry). In contrast, the following illustrates a function which fails definite unassignment:

```
function g(int x) -> int:
    final int y = 0
    //
    while x < 0:
        y = 1
        x = x + 1
    //
    return x + y
```

The above function fails definite unassignment because variable `y` is defined more than once when one or more iterations of the loop are executed (e.g. when `x<0` on entry). To illustrate the conservative nature of definite assignment, consider this variation:

```
function ten() -> int:
    int x = 0
    int y
    //
    while x < 10:
        y = 1
        x = x + 1
    //
    return y
```

Here, it can be shown that `y==1` must hold when the **`return`** statement is reached. Nevertheless, this function fails definite assignment because of the assumption that the loop executes *zero* or more iterations.

8.1.2 Infeasible Paths

Functions and methods may contain infeasible paths which are valid execution paths that, in practice, cannot be executed. The mechanism for checking definite (un)assignment

assumes for simplicity that any valid path can be executed. This means that some programs will fail definite assignment, even though they can be shown as safe. The following illustrates such a program:

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

This function contains four valid execution paths which can be denoted by `ff`, `tf`, `ft`, `tt` where, for example, `tf` represents the path where the first condition evaluates to `true` and the second to `false`. However, it is easy to see that execution paths `ff` and `tt` are infeasible. Furthermore, that on the other two paths, `tf` and `ft`, variable `y` is definitely assigned at the **`return`** statement. Despite this, the above function fails definite assignment because the mechanism considers all valid paths whilst ignoring infeasible execution paths.

8.1.3 Partial Assignments

A variable will never be considered definitely assigned after the application of one or more *partial assignments*. In contrast, a variable is no longer considered definitely unassigned in such case. The following illustrates a program which fails definite assignment even though it can be shown as safe:

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

function Point(int x, int y) -> Point:
    Point p
    p.x = x
    p.y = y
    return p
```

In the above program, the variable `p` can be shown as definitely assigned at the **`return`** statement. Nevertheless, the conservative mechanism for checking definite assignment will reject this program because variable `p` is initialised via partial assignments.

8.2 Description

Definite assignment is defined over the *control-flow graph* of a code block, such as a **`requires`** or **`ensures`** clause, or the body of a function or method. Appendix §**??** details the process for constructing a control-flow graph from a block of code.

Definite assignment considers the set of all *valid paths* in a control-flow graph. A valid path is a path through the graph starting from its root. Every vertex in the graph is associated with a set of variables which are *defined* at that vertex, as well as those which are *used* at that vertex. A variable x is said to be *definitely assigned* on entry to a vertex v if, for every valid path which includes v, some ancestor vertex u exists on which x is defined.

8.2.1 Definitions and Uses

A *variable definition* occurs at a vertex v which contains a direct assignment (recall §5.2.2), or an initialisation of (recall §5.2.6), one or more variables. Assignments to fields, array elements or dereferenced references are not direct and do not define variables. A vertex may define multiple variables if it corresponds to a multiple assignment which directly assigns those variables. Finally, the parameters of a function, method or invariant block are treated as being defined on entry.

A *variable use* occurs at a vertex v which directly refers to that variable in some expression other than an `LVal`. However, referring to a variable indirectly through a dereference expression does not constitute a variable use.

Chapter 9

Errors and Warnings

When the Whiley compiler encounters an invalid program it will report an *error*. In contrast, when it encounters something undesirable in an otherwise valid program, it may report a *warning*. This chapters details the complete list of error messages and warnings which can be reported for a Whiley program.

9.1 Overview

9.2 Parse Errors

9.3 Declarations

Declarations are top-level entities in a source file, and their syntax is defined in §3.

9.3.1 "Cyclic Constant Declaration" (E301)

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:

```
1 constant const1 is 1 + const1
2
3 constant const2 is 1 + const3
4 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`.

9.3.2 "Reference Not Permitted in Function" (E302)

A *reference not permitted in function* error occurs when an attempt is made to declare or use a variable of reference type in a function (as opposed to a method). Functions in Whiley must be free from side-effects — i.e. they must be *pure*. Thus, the potential side-effects made possible through the use of references is not permitted.

Example. The following illustrates a very simple example:

```
1 function f(&int x) -> (&int r):
2    return x
```

Here, function `f()` accepts a parameter `x` of reference type `&int`, which is not permitted. In this case the function does not, in fact, exhibit any side-effects; nevertheless, the function will currently be rejected.

9.3.3 "Reference Operation Not Permitted in Function" (E303)

A *reference operation not permitted in function* error occurs when an attempt is made to operate on a variable of reference type in a function (as opposed to a method). Functions in Whiley must be free from side-effects — i.e. they must be *pure*. Thus, the potential side-effects made possible through the use of references is not permitted.

Example. The following illustrates a very simple example:

```
1 function f(int x) -> (int r):
2    int y = x
3    return *(&y)
```

Here, function `f()` obtains a reference to local variable `y` and then immediately dereferences it, neither of which is permitted. In this case the function does not, in fact, exhibit any side-effects; nevertheless, the function will currently be rejected.

9.3.4 "Method Invocation Not Permitted In Function" (E304)

A *method invocation not permitted in function* error occurs when an attempt is made to call a method from a function (as opposed to another method). Functions in Whiley must be free from side-effects — i.e. they must be *pure*. Thus, the potential side-effects made possible through the method call are not permitted.

Example. The following illustrates a very simple example:

```
1 method g(int x) -> (int y):
2     return x
3
4 function f(int x) -> (int r):
5     return g(x)
```

Here, function `f()` accepts a parameter `x` and passes it through a call to method `g()`. In this case method `g()` does not, in fact, exhibit any side-effects; nevertheless, the method call will be rejected.

9.3.5 "Insufficient Return Values" (E305)

An *insufficient return values* error occurs when a **`return`** statement is encountered which does not provide as many return values as declared by the enclosing function or method.

Example. The following illustrates two simple examples:

```
1 method g(int x) -> (int y, int z):
2     return x+1
3
4 function f(int x) -> int:
5     return
```

Here, method `g()` is required to return *two* values but only one is actually being returned. Likewise, function `f()` is required to return one value but none are actually being returned.

9.3.6 "Too Many Return Values" (E306)

A *too many return values* error occurs when a **`return`** statement is encountered which provides more return values than declared by the enclosing function or method.

Example. The following illustrates two simple examples:

```
1 method g(int x):
2     return x
3
4 function f(int x) -> int:
5     return x,x+1
```

Here, method `g()` is required to return *zero* values but one is actually being returned. Likewise, function `f()` is required to return one value but two are actually being returned.

9.4 Types

9.4.1 "Subtype Error" (401)

A *subtype* error arises when an attempt is made to use an expression of type `T` in a position where an expression of type `S` is expected, and `T` is not a subtype of `S`. This is a common error precisely because it can occur in a large number of different situations.

Example. The following illustrates an example:

```
function f(int x) -> (int r):
    if x:
        return 0
    else:
        return 1
```

Here, variable `x` has type **`int`** but is being used in a position (i.e. as the condition of the **`if`**-statement) which expects a type **`bool`**. Another example is as follows:

```
function g(int[] items) -> (int r):
    return items
```

Here, variable `items` has type **`int`**`[]` but is being used in a position (i.e. as the return value) which expects a type **`int`**.

9.4.2 "Incomparable Operands" (402)

9.4.3 "Record Type Required" (403)

9.4.4 "Record Missing Field" (404)

9.5 Statements

Statements are used frequently in a Whiley program, and their syntax is defined elsewhere (see 5). The error messages reported in this section are those related to specific statement forms. Other, more general, errors can also be reported for a statement (e.g. *type errors*, §9.4) and are discussed elsewhere.

9.5.1 "Invalid LVal" (E501)

An *invalid lval* error occurs when an invalid expression is used on the left-hand side of an assignment. Only expressions which are also lval's maybe used in such a situation (see §5.2.2).

Example. The following illustrates two invalid lval's:

```
function f(int x):
    1 = x    // constant not valid lval
    x+1 = x // arithmetic expression not valid lval
```

The first assignment statement is invalid because one cannot assign to a constant. The second is invalid because one cannot assign to an arithmetic expression.

9.5.2 "Invalid Destructuring LVal" (E502)

An *invalid destructuring lval* error occurs when a destructuring assignment is used on the left-hand side, but the right-hand side returns an incorrect number of values.

Example. The following illustrates an invalid destructuring LVal:

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

function g(int x, int y) -> int:
    x,y = f(x,y)
    return x - y
```

Here, the invocation of `f()` in `g()` uses an invalid destructuring assignment because `f()` returns one value, but the assignment expects two.

9.5.3 "Variable Already Defined" (E503)

A *variable redefinition* error 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 sum(int[] items) => int[]:
    int i = 0
    int r = 0
    //
    while i < |items|:
       //
       int r = items[i]
       i = i + 1
    //
    return r
```

Here, the **while** loop attempts to declare a variable `r`, but another variable `r` was already declared beforehand.

9.5.4 "Unreachable Code" (E504)

An *unreachable statement* error arises in a function or method when no possible execution path could reach them.

Example. The following illustrates some unreachable code:

```
1 function abs(int x) -> int:
2     //
3     if x < 0:
4         return -x
5     else:
6         return x
7     //
8     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.

9.5.5 "Branch Always Taken" (E506)

A *branch always taken* error occurs when a conditional branch is determine to always evaluate to either `true` or `false`.

Example. The following illustrates a branch always taken:

```
1 function f(int x) -> int:
2    if x is int:
3        return x
4    else:
5        return -1
```

Here, the condition `x is int` always evaluates to `true` and, hence, the true branch of this conditional is always taken, whilst the false branch is never taken.

9.5.6 "Break Outside of Loop" (E507)

A *break outside loop* error occurs when a **break** statement is given which is not contained within one or more loops. This is an error because the break statement is used specifically to exit a loop early, and must be contained within the loop to be exited.

Example. The following illustrates a break outside of a loop:

```
function f(int x) -> int:
   break
   return x
```

Here, the **break** statement is meaningless as it is not associated with a loop.

9.5.7 "Duplicate Default Label" (E508)

A *duplicate default label* error 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.

9.5.8 "Duplicate Case Label" (E509)

A *duplicate case label* error 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.

9.6 Expressions

Expressions typically form the bulk of a Whiley program, and their syntax is defined elsewhere (see §6). The error messages reported in this section are those related to specific expression forms. More general errors can also be reported for an expression, such as *type errors* (see §9.4).

9.6.1 "Variable Possibly Uninitialised" (E601)

A *variable possibly uninitialised* error 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:

```
1 function f(int x) => int:
2    int y
3    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.

9.6.2 "Unknown Variable" (E602)

An *unknown variable* error occurs when an attempt is made to access a variable which has not been declared in the current scope. All variables must be declared before they can be used.

Example. The following illustrates an unknown variable:

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

Here, the **`return`** statements refers to an unknown variable `y`. In contrast, the reference to variable `x` is valid because `x` has been declared within scope.

9.6.3 "Unknown Function or Method" (E603)

An *unknown function or method* error occurs when an attempt is made to access a function or method which is not visible in the current scope. Functions and methods which are not declared in the same file as the invocation can be brought into the current scope using `import` statements.

Example. The following illustrates an invocation of an unknown function or method:

```
function f(int x) -> int:
   return g(x)
```

Here, function `g()` is not defined in the current source file and has not been brought into scope through an `import` statement.

9.6.4 "Ambiguous Coercion" (E604)

An *ambiguous coercion* error occurs when the target of a cast expression is uncertain. That is, when attempting to cast a value to a given type `T`, but there is more than one way this can be achieved. This error is reported as part of the *coercion check* performed during compilation.

Example. The following illustrates an ambiguous coercion:

```
type Ambiguous is { int f1, real f2} | { real f1, int f2 }

function f(int x, int y) -> Ambiguous:
   return (Ammbiguous) {f1: x, f2: y}
```

The cast is ambiguous here because it's unclear whether, for example, `{f1: 1, f2: 2}` should become `{f1: 1.0, f2: 2}` or `{f1: 1, f2: 2.0}`.

Glossary

access control Mechanisms for restricting the visibility of named declarations. 19

block comment A block comment begins with "/*" and continues until the end-of-comment marker "*/". 12

boolean expression An expression which evaluates to a value of type `bool`. 20, 38, 40, 45, 88

compilation group A group of one or more source files being compiled together. 17, 87

compilation unit A single unit of compilation. In Whiley, this includes source files and also binary WyIL files. 17–19

compile time The point in time at which a given compilation group is compiled into binary form.. 71, 77

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.. 18, 77

contractive A type is contractive if it does not describe an infinite series of self applications.. 21

control-flow graph A directed graph representation of a block of code (e.g. a function or method body) with which one can reason about the set of possible execution paths. 75, 88

declaration A declaration defines a new named entity within its enclosing source file.. 19, 20, 87

declaration modifier A declaration modifier provides additional meaning to a declaration.. 19

default package The top-level package which has no name, and is considered to be a "global" package.. 18

definite assignment Definite assignment is the process of checking that every variable is defined before being used.. 90

execution path A sequence of statements through a program, function or method which may be taken during execution.. 71, 72, 87

expression A combination of constants, variables and operators that, when evaluated, produce a single value. Expressions in certain circumstances may have side effects. 49, 87, 89, 90

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, 40, 53

foreign function interface A mechanism provided to enable inter-operation between Whiley source files and source files written in other languages.. 20

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.. 18

indentation syntax A lexical organisation of source files where indentation is significant and is used to group statements and blocks. 11

infeasible path A valid path through the control-flow graph of a function or method for which no valid parameter values exist which will let it be executed. 73

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. 25

line comment A line comment begins with "//" and continues until the end of line. 12

literal A source-level entity which describes a value of primitive type. 13

loop invariant A boolean expression which must hold on every iteration of a loop. 21, 23, 43, 47

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.. 18

name mangling The process of encoding information (e.g. about type parameters) within the exported name of a declaration.. 20

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. 19

negation type A type formed from another (e.g. `!int`), such that it includes any value not contained in the other. 25

overloading Overloading occurs when two entities in the same category exist with the same name, and is permitted only when their type allows for disambiguation.. 18

package A unit of hierarchical organisation within the Whiley namespace.. 17

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.. 21–23, 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.. 21–23, 32

refinable expression An expression which may be refined by a runtime type test. A refinable expression is either a variable access or a field access on a refinable expression. 69

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

side-effect A side-effect refers to the mutation of state that existed before a function or method was called, or the production of external effects through I/O. In Whiley, functions must be side-effect free, meaning they are not permitted to modify pre-existing state or interact through I/O. 21, 78

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.. 11, 17, 20–22, 32, 77, 87–90

statement An program instruction which has an effect on the environment when executed, but does not produce a value. 37, 89

statement block A sequence of zero or more consecutive statements with the same indentation. 12, 37, 45

type An abstract entity which represents the set of values a given variable may hold, or a given expression may evaluate to.. 25, 88–90

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.. 18

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. 25, 90

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. 25

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. 41, 90

variable definition A statement in which a variable is defined in its entirety, as opposed to a partial assignment of some part (e.g. field or array element). This concept is important in the process of checking *definite assignment*. 75

variable initialiser An optional expression used to initialise variable(s) declared as part of a variable declaration. 41

variable use A statement in which a variable is directly referred to in an expression other than an `LVal`. This concept is important in the process of checking definite assignment. 75

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. 17, 87

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 settheoretically with function, union, intersection, and negation types. *JACM*, 55(4):19:1–19:64, 2008.