The WHILE Language Specification

David J. Pearce and Lindsay Groves

March 11, 2016

Contents

Chapter 1

Introduction

The WHILE programming language is a simply imperative language which shares some similarity with C. Notable differences from C include the use of *structural subtyping* and the fact that WHILE is *strongly typed*. The WHILE language has been designed as a teaching aid which provides a useful (but not overly complex) feature set.

Several aspects of the language were chosen to simplify the WHILE compiler. Unlike many languages, WHILE programs are made up from only a single source file. This means the WHILE compiler does not have to be concerned with the complex issues related to *name resolution* and similar concerns. Similarly, the language does not support *overloading* of methods which simplifies the mechanism for deciding which method an invocation refers to.

1.1 Grammar notation

The syntax of the WHILE language is described using a form of extended BNF grammar, using the following conventions.

A grammar is a sequence of rules of the form lhs ::= rhs, where lhs is a *nonterminal symbol*, denoting a part of the language being defined, and rhs is a grammar expression describing the strings that are allowed as instances of lhs. A grammar expression can itself have one of several forms:

- A *terminal symbol*, or *terminal*, which appears as an actual symbol in a program. For clarity, to avoid confusipon with symbols used as part of the grammar, terminals are displayed in boxes, like this `X`. For example:

```
Colon ::= :
```

This says a `Colon` is a single symbol, `:`.

- A *non-terminal*, occuring on the left-hand side of some grammar rule. A non-terminal must occur on the left-hamd side of some grammar rule in order to be defined, and stands for any string withthe structure defined by that rule. Rules may be recursive, so a non-terminal may occur on the left- and right-hand sides of the same rule. For example:

```
VariableName ::= Identifier
```

  This says that a `VariableName` is any string that can be an `Identifier`.

- A *sequence* of grammar expressions. This describes a string made up of several substrings, with the structure describe by the grammar expressions in the sequence. For example:

```
DoubleColon ::= Colon Colon
```

  This says that a `DoubleColon` is a sequence of two `Colon`'s.

- A sequence of grammar expressions separated by *vertical bars*. These grammar expressions are understood as alternatives, so this describes a string which has the form described by one of the grammar expressions. For example:

```
Vowel ::= a | e | i | o | u
```

  This kind of rule is often defined as a short hand for several rules with the same left-hand side. However, it is good practice to collect all of the possible forms for a given nonterminal into a single rule like this.

- A grammar expression enclosed in *square brackets*. This describes an optional component.

```
ReturnStmt ::= return [Expr]
```

  This says that a return statement is the word **return**, optionally followed by an expression.

- A grammar expression to be *repeated*. A grammar expression followed by * may be repeated *zero or more times*. When followed my $^+$ it may be repeated *one or more times*. If the grammar expression to be repeated is more complex that a single terminal or non-terminal, we enclose if in parentheses. For example:

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

This says that an `Ident` is a letter followed by zero or more letters and/or digits. For example:

```
Word ::= Letter+
```

This says that a `Word` is one of more letters.

The grammar actually describes a larger language than the one we want, and additional constraints must be applied in order to exclude strings that are not actually part of the language. These include a number of context constraints (e.g. a method or variable cannot be declared twice in the same scope, the argument names of a method must be distinct, a method must be called with the same number of arguments as in its definition, methods and operators must be applied to arguments of the correct type, etc.). They also include constraints on the size of integer constants that can be used. Finally, in the grammar for expressions, the expressions given as arguments for any operator must not have a principal operator with lower precedence.[1] These constraints are not always stated explicitly!

[1] This rule is required because we simplify the grammar by always using `Expr` for the arguments, rather than building operator precedence into the grammar.

Chapter 2

Lexical Structure

This chapter specifies the lexical structure of the WHILE programming language. Programs in WHILE are organised into one or more *source file*s written in Unicode. The WHILE language uses curly braces to delimit blocks and statements, as found in languages like C or Java.

2.1 Line Terminators

A WHILE program consists of a sequence of ascii input characters, separated into lines by *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 Comments

There are two kinds of comments in While: *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.

2.3 Tokens and White Space

After comments have been removed, and While program consists of a sequence of *tokens*, as described in the rest of this section.

Every token must be entirely on one line, and consecutive tokens maybe separated by an arbitrary amount of *white space* (spaces and tabs). In some cases, white space is necessary in order to distinguish the tokens. For example, in `type Day is int`, the spaces are required, whereas in `2*(x-y)`, no spaces are needed.

In While, white space and line terminators are not significant, unlike Whiley or Python, for example, where line breaks and indentation are meaningful.

2.3.1 Identifiers

An identifier is a sequence of one or more *letters*, *digits* or underscores, starting with a letter or underscore.

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

_Letter ::= _ | Letter

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

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

A `Letter` is either a lowercase or uppercase alphabetic character (i.e. `a-z` and `A-Z`) underscore (_).

2.3.2 Keywords

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

```
Keyword ::= assert | bool | break | case | continue | default | else
          | false | for | if | int | is | print | return
          | skip | switch | true | void | while | const | type
```

2.3.3 Literals

A *literal* is a source-level entity which describes a value of primitive type (§4.3).

```
Literal ::= BoolLiteral
          | ByteLiteral
          | IntLiteral
          | CharacterLiteral
          | StringLiteral
```

2.3.3.1 Boolean Literals

The **`bool`** type (§4.3.1) has two values expressed as the `true` and `false` literals.

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

2.3.3.2 Integer Literals

An *integer literal* is a sequence of numeric digits (e.g. `123456`) corresponding to a value of **`int`** type (§4.3.2).

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

Integers are represented using 32bit *two's complement* notation, so integer literals must be between -2147483648 and 2147483647 (inclusive).

2.3.3.3 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 ascii code, which is necessary because there is no native character type.

```
CharacterLiteral ::= ' ( Character - ( \ | ' ) | CharacterEscape ) '

 CharacterEscape ::= \ ( \ | t | n | ' )

       Character ::= Letter | Digit | Symbol

       Character ::= Letter | Digit |

          Symbol ::= + | - | ...
```

`Symbol` represents all other printable ascii characters.

2.3.3.4 String Literals

A *string literal* is 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.

```
StringLiteral ::= " ( Character- ( \ | " ) | StringEscape )* "

 StringEscape ::= \ ( \ | t | n | " )
```

Chapter 3

Program Structure

A WHILE program is a sequence of data types and method declarations, contained in a single source file.

3.1 Programs

The syntax of a program is given as follows:

```
Program ::= (
          | TypeDecl
          | MethodDecl
          )*
```

3.2 Declarations

A *declaration* defines a new entity within a WHILE program and provides a *name* by which it can be referred to within this program.

3.2.1 Type Declarations

A *type declaration* declares a named type within a WHILE program. The declaration may refer to named types declared earlier in this program — for simplicity, it may not refer to itself (either directly or indirectly), so recursively defined types are not allowed.

```
TypeDecl ::= type Ident is Type
```

Examples. Some simple examples illustrating type declarations are:

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

This defines a record type `Point` which contains two integer fields, `x` and `y`.

3.2.2 Method Declarations

A *method declaration* defines a method within a WHILE program. Methods in WHILE are *impure* and may have side-effects. A method may call itself or other methods declared earlier in this program. Note that, for simplicity, mutually recursive methods are not allowed.

```
MethodDecl ::= Type Ident ( Parameters ) { Stmt* }

Parameters ::= [ Type Ident ( , Type Ident )* ]
```

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

```
1 int max(int x, int y) {
2    if(x > y) {
3       return x;
4    } else {
5       return y;
6    }
7 }
```

This implements the well-known *maximum* function for determining the larger of two integer parameters.

Chapter 4

Types & Values

The WHILE 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. WHILE's *type system* governs how the type of any variable or expression is determined. WHILE's type system is unusual in that it uses *structural typing*.

4.1 Overview

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

```
1 int|null id(null|int x) {
2     return x;
3 }
```

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 semantics (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 WHILE. The top-level grammar for type descriptors is:

```
     Type ::= UnionType
            | TermType

 TermType ::=
            | PrimitiveType
            | RecordType
            | ArrayType
            | ( Type )
```

4.3 Primitive Types

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

```
 PrimitiveType ::=
                | VoidType
                | BoolType
                | IntType
```

4.3.1 Booleans

The **`bool`** type represents the set of boolean values (i.e. `true` and `false`). Values of **`bool`** type support equality comparators (§6.5), binary logical operators (§6.7) and logical not (§6.7.1).

```
 BoolType ::= bool
```

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

```
1 bool contains(int[] list, int item) {
2     int i = 0;
3     while(i < |list|) {
4         if(list[i] == item) { return true; }
```

```
        i = i + 1;
    }
    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`.

4.3.2 Integers

The type **`int`** represents the set of integers representable in 32bits using *two's complement* notation, giving them a range of values between -2147483648 and 2147483647 (inclusive). Integer values are expressed as a sequence of one or more numerical digits (e.g. `123456`, etc). Values of **`int`** type support equality comparators (§6.5), relational comparators (§6.3.2), additive (§6.3.3), multiplicative (§6.3.4) and negation (§6.3.1) operations.

```
IntType ::= int
```

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

```
int fib(int x) {
    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.

4.3.3 Void

The type **`void`** is the empty set, and is used to represent the return type of a method which does not return anything. We cannot have variables of type **`void`**, because they cannot hold any possible value. Thus, **`void`** is the *bottom type* (i.e. $\bot$) in the lattice of types and, hence, is the *subtype* of all other types.

```
VoidType ::= void
```

Example. The following example illustrates the **`void`** type:

```
void helloWorld() {
   print "Hello World";
}
```

This illustrates the classical *Hello World* method which simply prints a short message to the console. Since the method does not return a value it has **`void`** return type.

4.4 Records

A record is a compound value made of one or more *fields*, each of which has a unique name and a corresponding type. Values of record type support equality comparators (§6.5) and field access (§6.8.1) operations, as well as field assignment (§5.1.2).

```
RecordType ::= { Type Ident ( , Type Ident )* }
```

Example. This example illustrates a record type:

```
type Rectangle is {
  int x, int y, int width, int height
}

int getArea(Rectangle r) { return r.width * r.height; }
```

This illustrates a record being used to define the concept of a `Rectangle`. A method is provided which, given a `Rectangle`, computes its area.

4.5 Arrays

An array is a sequence of values whose elements are subtypes of the element type. For example, `[1,2,3]` is an instance of array type **`int`**`[]`; however, `[false,true]` is not. Values of array type support equality comparators (§6.5) and access expressions (§6.4.2).

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

Example. The following example illustrates array types:

```
1 int[] add(int[] v1, int[] v2) {
2     int i=0;
3     while(i < |v1|) {
4         v1[i] = v1[i] + v2[i];
5         i = i + 1;
6     }
7     return v1;
8 }
```

This illustrates a method which adds each corresponding element from two integer arrays together.

4.6 Subtyping

Types in WHILE support the notion of *subtyping* where one type may be a *subtype* for another. For example, the type **`void`** is a subtype of all other types. Likewise, records support *depth* and *width* subtyping meaning that:

- `{int x, int y}` is a subtype of `{int x}` **(Width)**
- `{int[] xs}` is a subtype of `{void[] xs}` **(Depth)**

The *subtyping operator* is denoted by "$\leq$"; for example, $\mathtt{T}_1 \leq \mathtt{T}_2$ indicates that type $\mathtt{T}_1$ is a subtype of $\mathtt{T}_2$. The subtyping operator is *reflexive*, *transitive* and *anti-symmetric* with respect to the underlying types involved.

Chapter 5

Statements

The execution of a WHILE program is controlled by *statement*s, which cause effects on the environment. However, statements in WHILE do not produce values though they may result in I/O being performed. *Compound statements* may contain other statements.

5.1 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.

5.1.1 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.

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

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

```
1 int abs(int x) {
2     if(x < 0) {
3         x = -x;
4     }
5     assert x >= 0;
6     return x;
7 }
```

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.

5.1.2 Assignment Statement

An *assignment statement* is of the form `leftHandSide = rightHandSide`. Here, the `rightHandSide` is any expression, whilst the `leftHandSide` must be an `LVal` — that is, an expression which denotes a storage location into which a value can be stored. At runtime, the value generated by evaluating the right-hand side must be a subtype (§4.6) of that of the left-hand side.

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

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

Example. The following illustrates different possible assignment statements:

```
1  void f1(int[] x, int[] y) {
2      x = y;          // variable assignment
3  }
4
5  void f2({int f} x, int y) {
6      x.f = y;        // field assignment
7  }
8
9  void f3(int[] x, int i, int y) {
10     x[i] = y;       // list assignment
11 }
12
13 void f4({int f}[] x, int i, int y) {
14     x[i].f = y;     // compound assignment
15 }
```

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

5.1.3 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 (§**??**).

```
SkipStmt ::= skip
```

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

```
1  int abs(int x) {
2      //
3      if(x >= 0) {
4          skip;
5      } else {
6          x = -x;
7      }
8      //
9      return x;
10 }
```

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

5.1.4 Variable Declaration Statement

A *variable declaration* statement declares a variable that can be used within a method body. A variable declaration 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 [ = Expr ]
```

Example. Some example variable declarations are:

```
1  void f() {
2      int x;
3      int y = 1;
4      int z = y + y;
5  }
```

Here we see four variable declarations. The first has no initialiser, whilst the remainder have initialisers.

5.2 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.

5.2.1 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:

```
1  // find first element holding x from xs
2  int indexOf(int[] xs, int x) {
3      int i = 0;
4      while(i < |xs|) {
5          if(xs[i] == x) {
6          break;
7      } else {
8          i = i + 1;
9      }
10     }
11     return i;
12 }
```

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

5.2.2 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:

```
int sumNonNegative(int[] xs) {
    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.

5.3 For Statement

A for statement repeatedly executes a statement block so long as a given boolean expression (the condition) evaluates to `true`. A for statement includes an *initialiser* component and an *increment* component in addition to the loop condition.

```
ForStmt ::= for ( VariableDeclaration ; Expr ; Stmt ) { Stmt* }
```

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

```
int sum(int[] xs) {
    int r = 0;
    for(int i=0;i!=|xs|;i=i+1) {
       r = r + xs[i];
    }
    return r;
}
```

Here, we see a simple **`for`** statement which sums the elements of variable `xs`, storing the result in variable `r`.

5.3.1 If Statement

An **`if`** statement conditionally executes a *statement block* based on the outcome of a boolean expression. The boolean expression is referred to as the *condition*. The first block is referred to as the *true branch*, whilst the optional **`else`** block is referred to as

the *false branch*. **`if`** statements may be chained, so that one of several statement blocks (or none) is selected, based on a sequence of conditions.

```
IfStmt ::= if ( Expr ) { Stmt* }
           ( else if ( Expr ) { Stmt* } )*
           [ else { Stmt* } ]
```

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

```
int max(int x, int y) {
    if(x > y) {
        return x;
    } else if(x == y) {
        return 0;
    } else {
        return y;
    }
}
```

Here, we see an **`if`** statement with three possible outcomes, depending on the value of two conditions.

5.3.2 Return Statement

A *return statement* returns control to the caller of the enclosing method. It has an optional expression referred to as the *return value*, which if present is returned to the caller.

```
ReturnStmt ::= return [ Expr ]
```

A return value is required if the enclosing method has a non-void return type, but may be omitted if the return type is **`void`**.

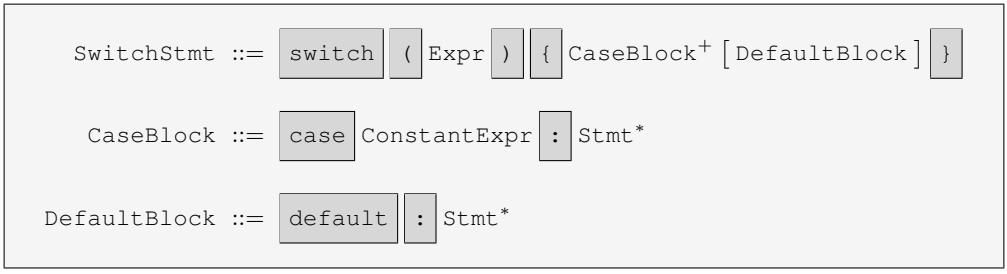
Example. The following illustrates a **`return`** statement:

```
int f(int x) {
    return x + 1;
}
```

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

5.3.3 Switch Statement

A *switch statement* executes 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, the **`switch`** statements executes the **`default`** block if one is given; otherwise, it does nothing, and control passes to the next statement following the **`switch`** statement.



Notice that the **`default`** block must come after the other cases, unlike Java where it can come any where. Also note that the values associated with the various cases must have the same type as the initial expression.

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

```
1 int sign(int x) {
2     switch(x) {
3         case 1:
4             return 1;
5         case -1:
6             return -1;
7     }
8     return 0:
9 }
```

Here, we see a simple **`switch`** statement which chooses between a number of possible values of **`int`** type.

5.3.4 While Statement

A while statement repeatedly executes a statement block so long as a given boolean expression (the condition) evaluates to `true`.

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

```
1 int sum(int[] xs) {
2     int r = 0;
3     int i = 0;
4     while(i < |xs|) {
5     r = r + xs[i];
6     i = i + 1;
7     }
8     return r;
9 }
```

Here, we see a simple **`while`** statement which sums the elements of variable `xs`, storing the result in variable `r`.

Chapter 6

Expressions

The majority of work performed by a While program is through the execution of *expressions*. Every expression produces a *value*, which may be stored in a variable or passed to a method.

6.1 Evaluation Order

Operators in While expressions are applied from left to right, subject to operator precedence and parentheses: operators with higher predence are applied before those with lower precednence, and operators within parentheses are applied before those outside of the parentheses. The operands of each operator are evaluated from left to right, and aside from the Boolean connectives (§6.7.2), operands are always fully evaluated before an 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*:

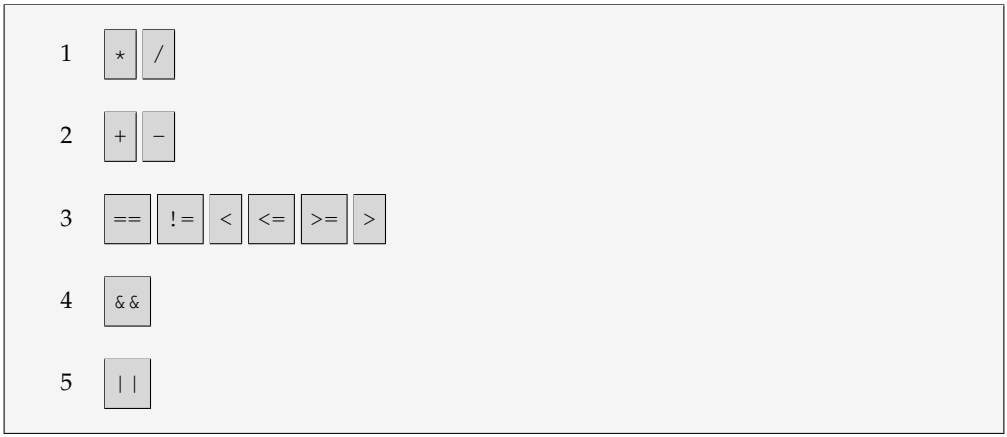
1. **Mixfix Expressions.** These operators take one or two operands with a non-infix syntax. This class includes the array access (§6.4.2) and field access operator (§6.8.1).

2. **Unary Expressions.** These operators take exactly one operand. This class takes highest precedence, and includes operators such as arithmetic negation (§6.3.1) and logical not (§6.7.1).

3. **Binary Expressions.** These operators take 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 (§6.7.2), etc.

4. **N-Ary Expressions.** These operators take an arbitrary number of operands. This class includes array constructors (§6.4.3), record constructors (§6.8.2), 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, and are evalauted before, higher ranked operators.

6.2 Expressions

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

```
Expr ::= ArithmeticExpr
       | EqualityExpr
       | InvokeExpr
       | LogicalExpr
       | ArrayExpr
       | RecordExpr
       | SimpleExpr
```

6.3 Arithmetic Expressions

Arithmetic expressions operate on values of integer type (i.e. **`int`**).

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

6.3.1 Negation Expressions

A negation expression takes one argument of integer 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 int negAccess(int i, int[] items) {
2    //
3    if(i < 0) {
4        return -items[-(i+1)];
5    } else {
6        return items[i];
7    }
8 }
```

6.3.2 Relational Expressions

A relational expression takes two arguments of integer type and performs an ordering comparison between then, returning a Boolean result.

```
ArithmeticRelationalExpr ::= Expr < Expr
                           | Expr <= Expr
                           | Expr => Expr
                           | Expr > Expr
```

Example. The following example illustrates the strict inequality comparators:

```
int compare(int x, int y) {
    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 takes two arguments of identical type (either **`int`**) and produces a result of matching type. The *addition operator*, `+`, adds both arguments together, whilst the *subtraction operator*, `-`, subtracts its right argument from its left argument.

```
ArithmeticAdditiveExpr ::= Expr ( + | - ) Expr
```

Example. The following illustrates the additive operators:

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

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

Example. The following illustrates the remainder operator:

```
int indexOf(int[] xs, int i) {
    //
    return xs[i % |xs|];
}
```

This function accepts a non-negative integer and uses this to index into an array. The remainder operator is used to ensure the array access is within bounds.

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|real)[]`, etc).

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

6.4.1 Length Expressions

The *lengthof operator* takes a value of array type, and returns 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
int firstOver(int[] items, int item) {
    int i = 0;
    while(i < |items|) {
        if(items[i] > item) {
            return item;
    }
        i = i + 1;
```

```
    }
    // no match
    return -1;
}
```

This 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 Array Access Expressions

An array access expression takes an array argument with one operand and returns the element at the given operand position in the array.

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

Arrays are indexed from zero. The value of the index expression must be be greater than or equal to zero and less than the length of the array; otherwise a fault is raised.

Examples. The following example illustrates the array access operator:

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

This 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 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 ::= [ ArgsList ]
```

Example. The following example illustrates an array initialiser:

```
1  // Create array of integers from start to end
2  int[] range(int start, int end) {
3      int[] r = [0;end-start];
4      int i = start;
5      //
6      while(i < end) {
7        r[i] = i;
8        i = i + 1;
9      }
10     //
11     return r;
12 }
```

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 Equality Expressions

An equality expression takes two arguments of the same type and compares them for equality or inequality, returning a boolean result.

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

Example. The following example illustrates an equality expression:

```
1  bool contains(int[] items, int item) {
2      //
3      int i = 0;
4      //
5      while(i < |items|) {
6          if(i == item) {
7              return true;
8      }
```

```
9           i = i + 1;
10      }
11      return false;
12  }
```

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.

6.5.1 Generator Expressions

An *array generator* takes 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:

```
1  int[] cons(int head, int[] tail) {
2      int[] r = [head; |tail| + 1];
3      int i = 0;
4      //
5      while(i < |tail|) {
6          r[i+1] = tail[i];
7          i = i + 1;
8      }
9      //
10     return r;
11 }
```

This function constructs an 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.

6.6 Invoke Expressions

A *method invocation* executes a named method declared in a given source file. An invocation passes arguments of appropriate number and type to the executed method. An invocation may also return a value which can be subsequently used.

```
InvokeExpr ::= Name ( ArgsList )

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

Example. The following example illustrates a function invocation:

```
1  // Determine the max of two values
2  int max(int x, int y) {
3      if(x >= y) {
4          return x;
5      } else {
6          return y;
7      }
8  }
9
10 // Determine the max of 1 or more values
11 int maxAll(int[] items) {
12     //
13     int r = 0;
14     int i = 0;
15     while(i < |items|) {
16         r = max(r,items[i]);
17         i = i + 1;
18     }
19     //
20     return r;
21 }
```

This example illustrates one function being called from another.

6.7 Logical Expressions

Logical expressions operate on values of **bool** type, and return a **bool** result.

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

6.7.1 Not Expressions

The *logical not* operator takes an argument of **bool** type and returns its logicazl negation.

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

Example. The following example illustrates the logical not operator:

```
int 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.

6.7.2 Connective Expressions

A logical connective takes two values of **bool** type (§4.3.1) and returns a **bool** value. The *logical AND* operator returns `true` if both operands are `true`, whilst the *logical OR* operator returns `true` if either operand is `true`.

```
LogicalBinaryExpr ::= Expr ( && | || ) Expr
```

Note that if the first argument of `&&` evaluates to `false`, or the first argument of `||` evaluates to `true`, then the second argument is not evaluated since it is not needed in order to determine the result. This is known as short-circuit evaluation.

This means that Boolean connectives in While (like most programming languages) are not commutative; i.e. E_1 `&&` E_2 and E_1 `||` E_2 are not necessarily equivalent to E_2 `&&` E_1 and E_2 `||` E_1, respectively. For example, when `k` is equal to `|A|`, `k < |A| && A[k] == x` returns `false`, whereas `A[k] == x && k < |A|` gives a fault.

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

```
bool implies(bool x, bool y) {
    return !x || y;
}

bool iff(bool x, bool y) {
    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.

6.8 Record Expressions

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

6.8.1 Field Access Expressions

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

```
FieldAccessExpr ::= Expr . Ident
```

In a field access `E.F`, the type of `E` must be a record type containing a field named `F`.

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

```
1 type Vec is {int x, int y, int z}
2
3 Vec dotProduct(Vec v1, Vec v2) {
4     return (v1.x * v2.x) + (v1.y * v2.y) + (v1.z * v2.z);
5 }
```

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.

6.8.2 Record Initialisers

A *record initialiser* takes one or more field names and expressions, and produces a value of record type in which the values of the expressions are associated with thbe corresponding field names.

```
RecordInitialiserExpr ::= { FieldArgsList }

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

Of course, field names must be distinct to ensure there is no ambiguity. Furthermore, no field is permitted to have **`void`** type.

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
Point move(Point p, int dx, int dy) {
    return { x: p.x+dx, y: p.y+dy };
}
```

This 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`.

6.9 Simple Expressions

A *simple expression* is either an atomic value (an identifier or a literal) or an expression enclosed in parentheses.

```
SimpleExpr ::= Ident
             | Literal
             | ( Expr )
```

Chapter 7

Type Checking

The WHILE 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. WHILE's *type system* governs how the type of any variable or expression is determined.

Chapter 8

Definite Assignment

The WHILE programming language requires that variables are known at *compile time* to be *definitely assigned* (i.e. that they are defined before used). A conservative approach is taken to determining whether or not this is the case. This ensures the language can be compiled efficiently, but also means that some provably safe programs are not valid WHILE programs. In this chapter, we specify the process by which definite assignment is determined.

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 int f(int x) {
2    int y;
3    //
4    if(x < 0) { y = 1; }
5    //
6    return x + y;
7 }
```

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. In fact, there are *two* possible execution paths through this function, but variable `y` 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.

8.1.1 Loops

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

```
1  int f(int x) {
2      int y;
3      //
4      while(x < 0) {
5          y = 1;
6          x = x + 1;
7      }
8      //
9      return x + y
10 }
```

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). To illustrate the conservative nature of the definite assignment mechanism, we can refined the above example as follows:

```
1  int ten() {
2      int x = 0;
3      int y;
4      //
5      while(x < 10) {
6          y = 1;
7          x = x + 1;
8      }
9      //
10     return y;
11 }
```

In this case, it can be shown that `y==1` must hold when the **`return`** statement is reached. Nevertheless, this function fails definite assignment because the mechanism naively assumes that the loop will execute *zero* or more iterations when, in fact, it will never execute zero 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 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:

```
int abs(int x) {
    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 the execution paths `ff` and `tt` are infeasible. Furthermore, we can see that on the other two paths, `tf` and `ft`, the variable `y` is definitely assigned when the **`return`** statement is reached. 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*. The following illustrates a program which fails definite assignment even though it can be shown as safe:

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

Point Point(int x, int y) {
    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.

Glossary

block comment A block comment begins with "/*" and continues until the end-of-comment marker "*/". 9

boolean expression An expression which evaluates to a value of type `bool`. 21, 25

compilation group A group of one or more source files being compiled together. 47

compile time The point in time at which a given *compilation group* is compiled into binary form. 43

compound statement A statement (e.g. `if`, `while`, etc) which may contain other statements. 21

control-flow graph A directed graph representation of a block of code (e.g. a method body) with which one can reason about the set of possible execution paths. 47

declaration A declaration defines a new named entity within a source file. 13

execution path A sequence of statements through a program or method which may be taken during execution. 43

expression A combination of constants, variables and operators that, when evaluated, produce a single value. 29, 47, 48

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. 21, 33

infeasible path A valid path through the *control-flow graph* of a method for which no valid parameter values exist which will let it be executed. 44

line comment A line comment begins with "//" and continues until the end of line. 9

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

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. 5

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. 5

source file A file in which source code is located. Source files for the While programming language have the extension `.while`. 9, 13, 47

statement An program instruction which has an effect on the environment when executed, but does not produce a value. 21, 48

statement block A sequence of zero or more consecutive statements. 25

type An abstract entity which represents the set of values a given variable may hold, or a given expression may evaluate to. 15, 48

type descriptor A source-level description of an underlying type. 15

value A value is an instance of a given type and permits a specific set of operations. Examples include: the integer value `1`; and the array value `[1,2]`. 29

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. 23, 48

variable initialiser An optional expression used to initialise variable(s) declared as part of a variable declaration. 23