The WHILE Language Specification

David J. Pearce

July 20, 2021

Contents

Chapter 1

Introduction

This document provides a specification of the WHILE programming language which is a simple, lightweight imperative language designed for teaching compilers. The intention of this document is to set out the syntax of the language and, to some extent, its semantics.

1.1 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 *non-terminal 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:

- **(Terminal)** A *terminal symbol*, or *terminal*, appears as an actual symbol in a program. To avoid confusion, terminals are displayed in boxes. For example:

```
Colon ::= :
```

  This indicates that `Colon` corresponds to a single symbol, `:`, in a program.

- **(Non-Terminal)** A *non-terminal symbol* must occur on the left-hand side of some grammar rule in order to be defined, and matches any string with the structure defined by that rule. For example:

```
VariableName ::= Identifier
```

  This says that a `VariableName` matches any string that can be an `Identifier`. Rules may be recursive, so a non-terminal may occur on the left- and right-hand sides of the same rule.

- **(Sequence)** A *sequence* of grammar expressions indicates a matching string requires matching several components in the order given. For example:

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

  This says that the (non-terminal) `DoubleColon` matches a sequence of exactly two `Colon`'s.

- **(Choice)** A *choice* of grammar expressions is indicated by separating them with vertical bars and indicates a matching string must match *one* of the options. For example:

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

This indicates that a `Vowel` may match any of the five alternative symbols. This kind of rule is often defined formally 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.

- **(Option)** A grammar expression enclosed in square brackets describes an *optional* component. For example:

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

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

- **(Repetition)** 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 than a single terminal or non-terminal, we enclose it 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. As another example:

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

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

1.2 Notes

The grammar in this document describes a larger language than might be desired, and additional constraints must be applied 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 methods 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, for example, 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 using `Expr` for the operands of arithmetic operators, rather than building operator precedence into the grammar.

Chapter 2

Lexical Structure

This chapter specifies the lexical structure of the WHILE programming language. 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. Note that block comments can span multiple lines, and cannot contain nested occurrences of `/*` or `*/`.

```
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

After comments have been removed, a 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 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, 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`).

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 | do | else
          | false | for | if | int | is | null | 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 (§4.2).

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

2.3.3.1 Null Literals

The **`null`** type (§4.2.1) has exactly one value expressed as the **`null`** literal.

```
NullLiteral ::= null
```

2.3.3.2 Boolean Literals

The **`bool`** type (§4.2.2) has two values expressed as the `true` and `false` literals.

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

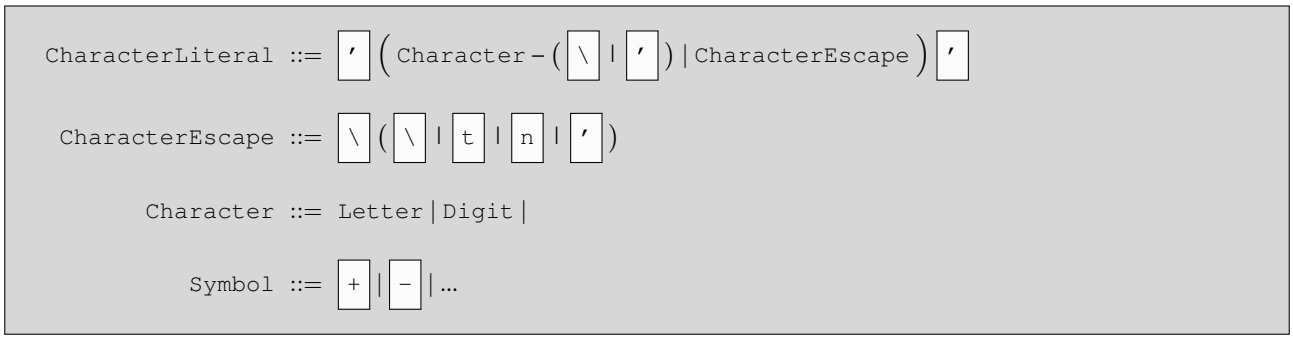
2.3.3.3 Integer Literals

An *integer literal* is a sequence of one or more numeric digits (e.g. `123456`) corresponding to a value of **`int`** type (§4.2.3).

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

2.3.3.4 Character Literals

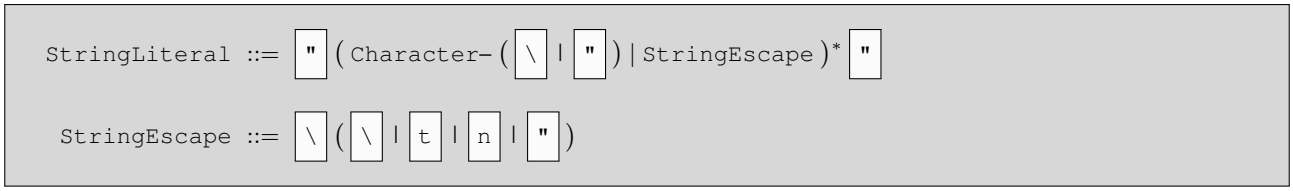
A *character literal* is expressed as a single character or an escape sequence enclosed in single quotes (e.g. `'c'`). Character literals generate integer constants corresponding to ASCII code, which is necessary because there is no native character type.



`Symbol` represents all other printable ASCII characters.

2.3.3.5 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 ASCII code points, which is necessary as there is no native string type. String literals must also appear entirely on one line.



Chapter 3

Program Structure

A WHILE program is a sequence of data types and method declarations, contained in a single *source file*. The choice to restrict programs to a single source file simplifies various issues, such as *name resolution*.

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
```

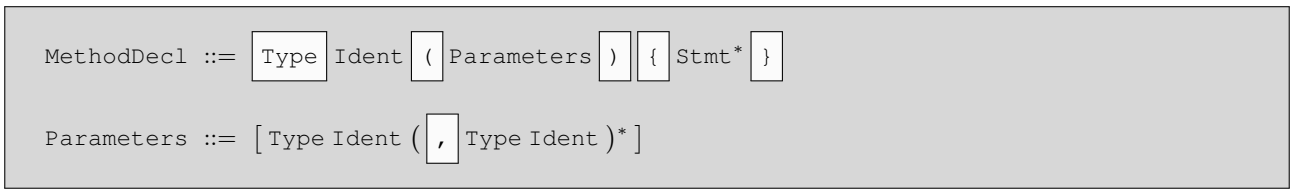
Example. The following illustrates a simple type declaration:

```
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.



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 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 ::= PrimitiveType
       | RecordType
       | ArrayType
       | ReferenceType
       | UnionType
```

4.2 Primitives

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

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

4.2.1 Null

The `null` type is used to show the absence of something. It is distinct from `void`, as 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 equality comparators (§6.5).

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

```
1 null Null() {
2    return null;
3 }
```

Although this method seems rather useless, it illustrates a key point: that `null` is both a type and a value in its own right.

4.2.2 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:

```
bool contains(int[] list, int item) {
   int i = 0;
   while i < |list| {
      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`.

4.2.3 Integers

The type `int` represents the set of integers representable in 32 bits 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.

4.2.4 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.

```
VoidType ::= void
```

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

```
void empty() {
   assert 1 == 1;
}
```

This illustrates a simple method which makes a trivial assertion that is always true. Since the method does not return a value it has `void` return type.

4.3 Records

A record is a compound value made of one or more *fields*, each of which has a unique name (within the record) 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 (§5.2.2).

```
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.

4.4 Arrays

An array is a sequence of values whose elements are of a given 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:

```
int[] add(int[] v1, int[] v2) {
   int i=0;
   while i < |v1| {
      v1[i] = v1[i] + v2[i];
      i = i + 1;
   }
   return v1;
}
```

This illustrates a method which adds each corresponding element from two integer arrays together. Note, nested arrays (e.g. `int[][]`) are also permitted.

4.5 References

A reference type is a *pointer* to a variable of a given element type which may, for example, be stored on the heap. For example, `new 1` returns an instance of reference type `&int` which has been freshly allocated on the heap. Values of reference type support equality comparators (§6.5).

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

Example. The following example illustrates reference types:

```
1 void main() {
2    &int r = new 1;
3    assert *r == 1;
4 }
```

4.6 Unions

A union type is constructed from two or more component types and may represent a value from any of its components. For example, the type `null|int` is a union which can be either an integer or null value. The set of values defined by a union type `T1|T2` is exactly the *union* of the sets defined by `T1` and `T2`. Values of union type support equality comparisons.

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

Example. The following example illustrates a union type:

```
1  // Return lowest index of matching item, or null if none
2  int|null indexOf(int[] items, int value) {
3     int i = 0;
4     while i < |items| {
5        if items[i] == value {
6           return i;
7        }
8        i = i + 1;
9     }
10    // item not found
11    return null;
12 }
```

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`).

Notes. Types in WHILE are challenging because of the distinction 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. The following illustrates 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 descriptors* expressed in the program text, namely `int|null` and `null|int`. Type descriptors occur at the source-level and describe types 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.

Since types are defined in terms of the 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`int | is equivalent to `int`
- `{int f,int g}`{int f} | is equivalent to `{int f}`{int f, int g} |
- `{int`null f} | is equivalent to `{int f}`{null f} |

Many programming languages support the concept of *subtyping* between types. Typically, for example, an assignment `x = y;` where `x` has type `T` and `y` has type `S` is permitted when `S` is a *subtype* of `T` (written $\texttt{S} \leq \texttt{T}$). Here, subtyping is used to indicate the assignment is safe (in some sense).

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. *Compound statements* may contain other statements.

5.1 Statements

```
Stmt ::= UnitStmt ; | BlockStmt
```

5.2 Unit Statements

A *unit statement* is a statement which does not contain other statements nested within them. Furthermore, in most (but not all) cases, control always continues to the next statement in sequence.

```
UnitStmt ::= AssertStmt
           | AssignStmt
           | BreakStmt
           | ContinueStmt
           | DeleteStmt
           | InvokeExpr
           | ReturnStmt
           | VarDecl
```

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.

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

Example. The following illustrates an `assert` statement:

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

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 that 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 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 type of the value generated by evaluating the right-hand side must match 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.

5.2.3 Break Statement

A *break statement* transfers control out of the lexically-nearest enclosing loop (i.e. `for`, `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 { break; }
6        i = i + 1;
7     }
8     return i;
9  }
```

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

5.2.4 Continue Statement

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

```
ContinueStmt ::= continue
```

Example. The following illustrates a `continue` statement:

```
1 int sumNonNegative(int[] xs) {
2    int r = 0;
3    for(int i=0;i<|xs|;i=i+1) {
4       if xs[i] < 0 { continue; }
5       r = r + xs[i];
6    }
7    return r;
8 }
```

Here, a `continue` statement ensures negative numbers are not included in the result of the function.

5.2.5 Delete Statement

A *delete statement* is of the form `delete e`, where `e` is a *reference expression*. Allocated memory referred to by the operand will be deallocated. A fault will be raised at runtime if the memory has already been deallocated.

```
DeleteStmt ::= delete Expr
```

Example. The following illustrates an `delete` statement:

```
1 void main() {
2    &int p = new 1;
3    assert *p == 1;
4    delete p;
5 }
```

Here, a reference to an integer is first allocated on the heap and then deleted.

5.2.6 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, and must be omitted if the return type is `void`.

Example. The following illustrates a `return` statement:

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

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

5.2.7 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     int[] xs = [1,2,3];
6     {int f} rec = {f:1};
7  }
```

5.3 Block Statements

A *block statement* is a compound statement which may contain nested statements and may have multiple exit points.

```
BlockStmt ::= DoWhileStmt | ForStmt | IfStmt | SwitchStmt | WhileStmt
```

5.3.1 Do-While Statement

A do-while statement executes a *statement block* at least once, and then repeatedly whilst a given boolean expression (the *condition*) evaluates to `true`.

```
DoWhileStmt ::= do { Stmt* } while Expr ;
```

Example. The following illustrates a do-while statement:

```
1  bool allPositive(int[] items) {
2    int i = 0;
3
4    do {
5      if items[i] < 0 { return false; }
6      i = i + 1;
7    } while i < |items|;
8
9    return true;
10 }
```

Here, we see a simple do-while statement which checks whether the elements of a given array are all positive or not. Note this program assumes the array has at least one element on entry!

5.3.2 For Statement

A `for` statement repeatedly executes a *statement block* so long as a given boolean expression (the *condition*) evaluates to `true` and, additionally, supports the declaration and increment of a loop index variable.

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

Example. The following illustrates an `for` statement:

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

Here, we see a `for` statement being used to sum an array of integers.

5.3.3 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.

5.3.4 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.

```
SwitchStmt ::= switch Expr { CaseBlock+ [ DefaultBlock ] }

CaseBlock ::= case ConstantExpr : Stmt*

DefaultBlock ::= default : Stmt*
```

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:

```
int[] toDescriptorString(int t) {
   switch t {
      case 0:
         return "Red";
      case 1:
         return "Green";
      case 2:
         return "Blue";
      default:
         return "Unknown";
   }
}
```

Here, we see a simple `switch` statement which chooses between a number of possible integer values. A `default` case is given which catches all remaining cases.

5.3.5 While Statement

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

```
WhileStmt ::= while Expr { Stmt* }
```

Example. The following illustrates an `while` statement:

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

Here, a simple `while` statement 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 precedence, 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 parentheses, a fixed *operator precedence* is used. This is first determined by *operator class*:

1. **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).
2. **Binary (Infix) 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.
3. **Binary (Mixfix) Expressions.** These operators take two operands but are non-infix operators and, hence, precedence is not ambiguous. This class includes the array access (§6.4.2) operator.
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:

3	*Operators*	`+` `-` `*` `/`
2	*Comparators*	`==` `!=` `<` `<=` `>=` `>`
1	*Connectives*	`&&` `||`

Higher ranked operators bind more tightly (i.e. take higher precedence) than, and are evaluated before, lower ranked operators. Furthermore, expressions containing operators at the same level are not permitted to be ambiguous and, instead, explicit bracing is required. For example, `x + 2 * y` is not permitted, whilst `x + (2 * y)` is.

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
       | ArrayExpr
       | EqualityExpr
       | InvokeExpr
       | LogicalExpr
       | RecordExpr
       | ReferenceExpr
       | SimpleExpr
       | TypeExpr
```

6.3 Arithmetic Expressions

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

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

6.3.1 Negation Expressions

A negation expression takes 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 int negAccess(int i, int[] items) {
2    if i < 0 { return -items[-(i+1)]; }
3    return items[i];
4 }
```

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 integer type 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 integer type 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[]`, etc).

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

6.4.1 Array 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 items[i]; }
      i = i + 1;
   }
   // no match
   return item;
}
```

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 a *source operand* of array type and an *index operand* of integer type 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 int[] mkpair(int first, int second) { return [1,2]; }
```

This example uses an array initialiser to construct an array of length two containing both parameters.

6.4.4 Array Generator Expressions

An *array generator* takes two operands and produces a value of array type. The first operand can be of any type, whilst the second operand must be of integer 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[] append(int[] items, int item) {
2    int[] r = [item; |items|+1];
3    //
4    for(int i=0;i!=|items|;i=i+1) {
5       r[i] = items[i];
6    }
7    return r;
8 }
```

The above method appends a given item onto the end of an array. The array generator is used to construct a new array of size one greater than the original.

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    int i = 0;
3    while i < |items| {
4       if i == item { return true; }
5       i = i + 1;
6    }
7    return false;
8 }
```

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.

Notes. Equality and inequality are defined for values of *any* type, including arrays and records.

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. An invocation may be used as an expression, in which case it *must* return a value. Alternative, it may be used as a statement, in which case it *may* return a value (and, if so, this is discarded).

```
InvokeExpr ::= Ident ( ArgsList )

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

Example. The following example illustrates a function invocation:

```
1  int max(int x, int y) {
2     if x >= y { return x; }
3     else { return y; }
4  }
5
6  // Determine the max of 1 or more values
7  int max_n(int[] items) {
8     int r = items[0];
9     //
10    for(int i=1;i!=|items|;i=i+1) {
11       r = max(r,items[i]);
12    }
13    return r;
14 }
```

This example illustrates one method (`max()`) being called from another (`max_n()`). Observe that methods are not permitted to have the same name (i.e. *overloading* is not supported).

6.7 Logical Expressions

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

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

6.7.1 Not Expressions

The *logical not* operator takes an argument of `bool` type and returns its logical 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 (§4.2.2) 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. `e1 && e2` and `e1 || e2` are not necessarily equivalent to `e2 && e1` and `e2 || e1`, respectively. For example, when `k` is equal to `|A|`, `k < |A| && A[k] == x` returns `false`, whereas `A[k] == x && k < |A|` raises 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).

```
RecordExpr ::= FieldAccessExpr | RecordInitialiserExpr
```

6.8.1 Field Access Expressions

The field access operator takes a value of record type along with a field name and returns the value held in that 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:

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

Vec dotProduct(Vec v1, Vec v2) {
   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.

6.8.2 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 the corresponding field names.

```
RecordInitialiserExpr ::= { FieldArgsList }

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

Of course, the field names must be distinct.

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 Reference Expressions

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

```
ReferenceExpr ::= DereferenceExpr
                | FieldDereferenceExpr
                | NewExpr
```

6.9.1 Dereference Expressions

The dereference operator takes a reference and returns the value held in the variable to which it refers.

```
DereferenceExpr ::= * Expr
```

Examples. The following example illustrates a field dereference expression:

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

void main() {
   &Point r = new {x:1,y:2};
   //
   assert r->x == 2;
   assert r->y == 3;
}
```

6.9.2 Field Dereference Expressions

The field dereference operator takes a reference to a record along with a field name and returns the value held in that field.

```
FieldDereferenceExpr ::= Expr [->] Ident
```

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

Examples. The following example illustrates a dereference expression:

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

int get_x(&Point p) {
   return p->x;
}
```

6.9.3 New Expressions

The new operator allocates space in the heap for a given value, and returns a reference to it.

```
NewExpr ::= [new] Expr
```

Examples. The following example illustrates a new expression:

```
void main() {
   &int p = new 1;
   //
   assert *p == 1;
}
```

6.10 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 [)]
```

6.11 Type Expressions

Type expressions operate on the type of a given expression.

```
TypeExpr ::= CastExpr
           | IsExpr
```

6.11.1 Cast Expressions

A *cast operator* accepts a value of one type $\mathtt{T}_1$ and returns a value of a different type $\mathtt{T}_2$, where $\mathtt{T}_1 \leq \mathtt{T}_2$ or $\mathtt{T}_2 \leq \mathtt{T}_1$. This may result in a change of the underlying representation.

```
CastExpr ::= ( Type ) Expr
```

Example. The following illustrates a cast operator being used:

```
1 int extract(int|null x) {
2    if x != null { return (int) x;
3    } else { return 0; }
4 }
```

This examines a value of union type to see whether or not it contains the `null` value and, if not, returns the integer value, otherwise it returns a default value.

6.11.2 Is Expressions

The *type test operator*, `is`, is used to determine the type of a value stored in a union. The syntax for this operator is:

```
IsExpr ::= Expr is Type
```

Observe that, for a type test `e is S` where `e` has type `T` we require that $\mathtt{S} \leq \mathtt{T}$. Thus, for example, `x is null` yields a syntax error when `x` has type `int|bool`.

Example. The following illustrates a type test operator being used:

```
1 type msg1 is {int kind, int item}
2 type msg2 is {int kind, int[] items}
3
4 int|int[] getData(msg1|msg2 m) {
5    if m is msg1 { return m.item; }
6    else { return m.items; }
```

Here, the runtime type test, `is`, is used to determine what type of value is stored in `m`.

Glossary

block comment A block comment begins with "/*" and continues until the end-of-comment marker "`*/`". 7

boolean expression An expression which evaluates to a value of type **`bool`**. 19, 23

compound statement A statement (e.g. **`if`**, **`while`**, etc) which may contain other statements. 19

declaration A declaration defines a new named entity within a source file. 11

expression A combination of constants, variables and operators that, when evaluated, produces a single value. 25, 35

fault A fault is raised when an unrecoverable error in the program occurs. 19, 21, 27

line comment A line comment begins with "//" and continues until the end of line. 7

literal A source-level entity which describes a value of primitive type. 8

reference expression An expression which evaluates to a value of reference type. 21

source file A file in which source code is located. Source files for the While programming language have the extension **`.while`**. In While, source files can be executed directly using the interpreter, or compiled into a binary form. 11, 35

statement An program instruction which has an effect on the environment when executed, but does not produce a value. 19, 35

statement block A sequence of zero or more consecutive statements. 23

type An abstract entity which represents the set of values a given variable may hold, or a given expression may evaluate to. 35

value A value is an instance of a given type and permits a specific set of operations. Examples include: the integer value `1`; the array value `[1,2]`; and the record value `{f:1}`. 25

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. 22, 35

variable initialiser An optional expression used to initialise variable(s) declared as part of a variable declaration. 22