The Wool Reference Manual[*]

Gary Pollice

v 1.6
February 10, 2021

Contents

1 Introduction **3**

2 Getting Started **3**

3 Classes **3**
3.1 Features 4
3.2 Inheritance 5

4 Types **5**
4.1 Type Checking 6

5 Attributes **6**
5.1 Default initialization 6

6 Methods **7**

7 Expressions **7**
7.1 Constants 8
7.2 Identifiers 8
7.3 Assignment 8
7.4 Dispatch 8
7.5 Conditionals 9
7.6 Loops 9
7.7 Blocks 9
7.8 Selection 10
7.9 new 10
7.10 isnull 10
7.11 Arithmetic and Comparison Operations 10

[*]Copyright ©1995-2004 by Alex Aiken. Modifications copyright ©2017-2021 by Gary Pollice. All rights reserved.

8 Basic Classes **11**
8.1 Object . . . 11
8.2 IO . . . 11
8.3 Str . . . 12
8.4 Primitives . . . 12

9 Main Method **12**

10 Lexical Structure **13**
10.1 Integers, Identifiers, and Special Notation . . . 13
10.2 Strings . . . 13
10.3 Comments . . . 13
10.4 Keywords . . . 13
10.5 White Space . . . 13
10.6 A small ambiguity . . . 14

11 Wool Syntax **15**
11.1 Precedence . . . 16

12 Implementation Details **16**
12.1 The Wool Namespace . . . 16
12.2 Source files . . . 17
12.3 The Woolc tool . . . 17
12.3.1 woolc man page . . . 18
12.4 Running Wool programs . . . 19

13 Acknowledgements **19**

1 Introduction

This manual describes the programming language Wool–U . Wool–U is a language that is designed for my undergraduate CS4533: Techniques of Programming Language Translation course. This is basically an undergraduate compilers course. There is a gradate course CS544: Compiler Construction, that is offered at the same time as CS4533. CS544 has a separate reference manual for the Wool–G languages. Wool–G extends Wool–U with additional types and semantics, as well as requiring optimization. I will use the "Wool " and Wool–U interchangeably in this document and use Wool–G only if I want to emphasize a feature of Wool–G that is not in Wool–U .

Wool is derived from existing commercial languages, and several languages that have been designed for courses at other institutions. Specifically, I make heavy use of Cool: the *C*lassroom *O*bject-*O*riented *L*anguage originally developed by Alex Aiken at Stanford University, Decaf and Espresso designed for courses at MIT, and Espresso from Columbia University. Wool stands for WPI Object-Oriented Language. It is a small language that can be implemented without optimizations with reasonable effort in a 7-week It retains many of the features of modern programming languages including static typing, objects, and lexical scoping.

Wool programs are sets of *classes*. A class encapsulates the variables and methods of a data type. Instances of a class are *objects*. In Wool, classes and types are identified; i.e., every class defines a type. Classes permit programmers to define new types and associated procedures (or *methods*) specific to those types. Inheritance allows new types to extend the behavior of existing types.

Wool is an *expression* language. Most Wool constructs are expressions, and every expression has a value and a type. Wool is *type safe*: procedures are guaranteed to be applied to data of the correct type. While static typing imposes a strong discipline on programming in Wool, it guarantees that no run time type errors can arise in the execution of Wool programs.

2 Getting Started

See the course Web pages and lectures for information on beginning to work with Wool and the development environment.

3 Classes

All code in Wool is organized into classes. Each class definition must be contained in a single source file, but multiple classes may be defined in the same file. Class definitions have the form:

```
class <type> [ inherits <type> ] {
    <feature_list>
}
```

Square brackets like this `[...]` denote an optional construct. Identifiers in angle brackets indicate a generic name that must be specified: `<name>`. All class names are globally visible. Class names begin with an uppercase letter. Classes may not be redefined.

You will be provided with code that will compiles Wool source files by concatenating them into a single text stream that can be compiled in a single pass. This allows you to write, as you might in Java, one class per file and have several classes used in an application. It allows us to bypass the `import` mechanism of Java or other similar approaches.

3.1 Features

The body of a class definition consists of a list of feature definitions. A feature is either an *attribute* or a *method*. An attribute of class A specifies a variable that is part of the state of object instances of class A. A method of class A is a procedure that may manipulate the variables and object instances of class A.

One of the major themes of modern programming languages is *information hiding*, which is the idea that certain aspects of a data type's implementation should be abstract and hidden from users of the data type. Wool supports information hiding through a simple mechanism: all attributes have scope local to their class (i.e. they are like private Java and C++ instance variables), and all methods have global scope. Thus, the only way to provide access to object state in Wool is through methods.

Feature names *must begin with a lowercase letter.* No method name may be defined multiple times in a class, and no attribute name may be defined multiple times in a class hierarchy or its sub-classes, but a method and an attribute may have the same name. All methods in a class hierarchy that share the same name must have the same signatures; that is, the number, type, and order of their parameters must be identical, and the return types must be identical.

A fragment from list.wl illustrates simple cases of both attributes and methods:

```
class Cons inherits List {
        xcar : int;
        xcdr : List;

        isNil() : boolean { false }

        init(hd : int, tl : List) : Cons {
          {
            xcar <- hd;
            xcdr <- tl;
            this;
          }
        }
...
}
```

In this example, the class `Cons` has two attributes `xcar` and `xcdr` and two methods `isNil()` and `init()`. Note that the types of attributes, as well as the types of formal parameters and return types of methods, are explicitly declared by the programmer.

Given object `c` of class `Cons` and object `l` of class `List`. we can set the `xcar` and `xcdr` fields by using the `init()` method:

```
c.init(1,l)
```

This notation is *object-oriented dispatch.* There may be many definitions of `init()` methods in many different classes. The dispatch looks up the class of the object `c` to decide which `init()` method to invoke. Because the class of `c` is `Cons`, the `init()` method in the `Cons` class is invoked. Within the invocation, the variables `xcar` and `xcdr` refer to `c`'s attributes. The special variable `this` refers to the object on which the method was dispatched, which, in the example, is `c` itself.

There is a special form `new C` that generates a fresh object of class `C`. An object can be thought of as a record that has a slot for each of the attributes of the class as well as pointers to the methods of the class. A typical dispatch for the `init()` method is:

```
(new Cons).init(1,new Nil)
```

This example creates a new cons cell and initializes the "car" of the cons cell to be `1` and the "cdr" to be `new Nil`.[1] There is no mechanism in Cool–W for programmers to deallocate objects.

Attributes are discussed further in Section 5 and methods are discussed further in Section 6.

3.2 Inheritance

If a class definition has the form

```
class C inherits P { ... }
```

then class `C` inherits the features of `P`. In this case `P` is the *parent* class of `C` and `C` is a *child* class of `P`.

The semantics of `C inherits P` is that `C` has all of the features defined in `P` in addition to its own features. In the case that a parent and child both define the same method name, then the definition given in the child class takes precedence. *It is illegal to redefine attribute names.* Furthermore, for type safety, it is necessary to place some restrictions on how methods may be redefined (see Section 6).

There is a distinguished class `Object`. If a class definition does not specify a parent class, then the class inherits from `Object` by default. A class may inherit only from a single class; this is aptly called "single inheritance."[2] The parent-child relation on classes defines a graph. This graph may not contain cycles. For example, if`C` inherits from `P`, then `P` must not inherit from `C`. Furthermore, if `C` inherits from `P`, then `P` must have a class definition somewhere in the program. Because Cool–W has single inheritance, it follows that if both of these restrictions are satisfied, then the inheritance graph forms a tree with `Object` as the root.

In addition to `Object`, Cool–W has two other *basic classes*: `Str`, and `IO`. It also has the primitive "classes" `int` and `boolean` that have no features and correspond to the Java primitives of the same name.

The basic classes are discussed in Section 8.

4 Types

In Cool–W, every class name is also a type. A *type declaration* has the form `x:C`, where `x` is a variable and `C` is a type. Every variable must have a type declaration at the point it is introduced. The types of all attributes must be declared.

The basic type rule in Wool is that if a method or variable expects a value of type `P`, then any value of type `C` may be used instead, provided that `P` is an ancestor of `C` in the class hierarchy. In other words, if `C` inherits from `P`, either directly or indirectly, then a `C` can be used wherever a `P` would suffice.

When an object of class `C` may be used in place of an object of class `P`, we say that `C` *conforms* to `P` or that $\mathtt{C} \leq \mathtt{P}$ (think: `C` is lower down in the inheritance tree). Conformance is defined in terms of the inheritance graph.

Definition 4.1 (Conformance) Let $\mathtt{A}, \mathtt{C}$, and `P` be types.

- $\mathtt{A} \leq \mathtt{A}$ for all types `A`
- if `C` inherits from `P`, then $\mathtt{C} \leq \mathtt{P}$

[1] In this example, `Nil` is assumed to be a subtype of `List`.

[2] Some object-oriented languages allow a class to inherit from multiple classes, which is equally aptly called "multiple inheritance."

- if $\mathtt{A} \leq \mathtt{C}$ and $\mathtt{C} \leq \mathtt{P}$ then $\mathtt{A} \leq \mathtt{P}$

Because `Object` is the root of the class hierarchy, it follows that $\mathtt{A} \leq \mathtt{Object}$ for all types `A`.

4.1 Type Checking

The Wool type system guarantees at compile time that execution of a program cannot result in run time type errors. Using the type declarations for identifiers supplied by the programmer, the type checker infers a type for every expression in the program.

It is important to distinguish between the type assigned by the type checker to an expression at compile time, which we shall call the *static* type of the expression, and the type(s) to which the expression may evaluate during execution, which we shall call the *dynamic* types.

The distinction between static and dynamic types is needed because the type checker cannot, at compile time, have perfect information about what values will be computed at runtime. Thus, in general, the static and dynamic types may be different. What we require, however, is that the type checker's static types be *sound* with respect to the dynamic types.

Definition 4.2 For any expression `e`, let $\mathtt{D}_\mathtt{e}$ be a dynamic type of `e` and let $\mathtt{S}_\mathtt{e}$ be the static type inferred by the type checker. Then the type checker is *sound* if for all expressions `e` it is the case that $\mathtt{D}_\mathtt{e} \leq \mathtt{S}_\mathtt{e}$.

Put another way, we require that the type checker err on the side of overestimating the type of an expression in those cases where perfect accuracy is not possible. Such a type checker will never accept a program that contains type errors. However, the price paid is that the type checker will reject some programs that would actually execute without run time errors.

5 Attributes

An attribute definition has the form

```
<id> : <type> [ <- <expr> ];
```

The expression is optional initialization that is executed when a new object is created. The static type of the expression must conform to the declared type of the attribute. If no initialization is supplied, then the default initialization is used (see below).

When a new object of a class is created, all of the inherited and local attributes must be initialized. Inherited attributes are initialized first in inheritance order beginning with the attributes of the greatest ancestor class.[3] Within a given class, attributes are initialized in the order they appear in the source text.

Attributes are local to the class in which they are defined or inherited. Inherited attributes cannot be redefined.

5.1 Default initialization

All variables in Wool are initialized to contain values of the appropriate type. The special value `null` is a member of all non-primitive types and is used as the default initialization for class variables when

[3]That is, the class that is highest in the inheritance hierarchy. Thhe `Object` class is always the highest (greatest ancestor) from any class.

no other initialization is specified, if allowed in the language. This is the same as `null` in Java. The primitive types have the same initialization as in Java: `int` is initialized to 0 and `boolean` to `false`.

There is a special form, `isnull` that tests whether a value is `null` (see Section 7.10). In addition, `null` values may be tested for equality. A `null` value may be passed as an argument, assigned to a class variable, or otherwise used in any context where any class value is legitimate, except that a dispatch to or case on `null` generates a run time error.

Variables of the basic class `Str` are initialized specially; see Section 8.

6 Methods

A method definition has the form

```
<id>(<id> : <type>,...,<id> : <type>): <type> { <vardef>* <expr> }
```

There may be zero or more formal parameters. The identifiers used in the formal parameter list must be distinct. The type of the method body must conform to the declared return type. When a method is invoked, the formal parameters are bound to the actual arguments and the expression is evaluated; the resulting value is the meaning of the method invocation. *A formal parameter hides any definition of an attribute of the same name.*

To ensure type safety, there are restrictions on the redefinition of inherited methods. The rule is simple: If a class `C` inherits a method `f` from an ancestor class `P`, then `C` may override the inherited definition of `f` provided the number of arguments, the types of the formal parameters, and the return type are exactly the same in both definitions.

To see why some restriction is necessary on the redefinition of inherited methods, consider the following example:

```
class P {
   f(): int { 1 }
}

class C inherits P {
   f(): Str { "1" }
}
```

Let `p` be an object with dynamic type `P`. Then

```
p.f() + 1
```

is a well-formed expression with value 2. However, we cannot substitute a value of type `C` for `p`, as it would result in adding a string to a number. Thus, if methods can be redefined arbitrarily, then subclasses may not simply extend the behavior of their parents, and much of the usefulness of inheritance, as well as type safety, is lost.

7 Expressions

Expressions are the largest syntactic category in Wool.

7.1 Constants

The simplest expressions are constants. The `boolean` constants are `true` and `false`. The `int` constants are unsigned strings of digits such as `0`, `123`, and `007`. `Str` constants are sequences of characters enclosed in double quotes, such as `"This is a string."` The value of a `Str` constant is an object of the `Str` class.

7.2 Identifiers

The names of local variables, formal parameters of methods, `this`, and class attributes are all expressions. The identifier `this` may be referenced, but it is an error to assign to `this`. It is also illegal to have attributes named `this`.

Local variables and formal parameters have lexical scope. Attributes are visible throughout a class in which they are declared or inherited, although they may be hidden by local declarations within expressions. The binding of an identifier reference is the innermost scope that contains a declaration for that identifier, or to the attribute of the same name if there is no other declaration. The exception to this rule is the identifier `this`, which is implicitly bound in every class.

7.3 Assignment

An assignment has the form

```
<id> <- <expr>
```

The static type of the expression must conform to the declared type of the identifier. The value is the value of the expression. *The static type of an assignment is the static type of* `<expr>`.

7.4 Dispatch

There are two forms of dispatch (i.e. method call) in Wool. The forms differ only in how the called method is selected. The most commonly used form of dispatch is

```
<expr>.<id>(<expr>,...,<expr>)
```

Consider the dispatch e_0.`f`($e_1, \ldots, e_n$). To evaluate this expression, the arguments are evaluated in left-to-right order, from e_1 to e_n. Next, e_0 is evaluated and its class `C` noted (if e_0 is `null` a run time error is generated). Finally, the method `f` in class `C` is invoked, with the value of e_0 bound to `this` in the body of `f` and the actual arguments bound to the formals as usual. The value of the expression is the value returned by the method invocation.

Type checking a dispatch involves several steps. Assume e_0 has static type `A`. (Recall that this type is not necessarily the same as the type `C` above. `A` is the type inferred by the type checker; `C` is the class of the object computed at run time, which is potentially any subclass of `A`.) Class `A` must have a method `f`, the dispatch and the definition of `f` must have the same number of arguments, and the static type of the ith actual parameter must conform to the declared type of the ith formal parameter. If `f` has return type `B` and `B` is a class name, then the static type of the dispatch is `B`.

The other form of dispatch is:

```
<id>(<expr>,...,<expr>)
```

This is shorthand for `this.<id>(<expr>,...,<expr>)`.

7.5 Conditionals

A conditional has the form

```
if <expr> then <expr> else <expr> fi
```

The semantics of conditionals is standard. The predicate is evaluated first. If the predicate is `true`, then the `then` branch is evaluated. If the predicate is `false`, then the `else` branch is evaluated. The value of the conditional is the value of the evaluated branch.

The predicate must have static type `boolean`. The branches may have any static types. To specify the static type of the conditional, we define an operation $\sqcup$ (pronounced "join") on types as follows. Let `A`,`B`,`D` be any type. The *least type* of a set of types means the least element with respect to the conformance relation $\leq$.

$$\begin{array}{rcll}
\mathtt{A} \sqcup \mathtt{B} & = & \text{the least type } \mathtt{C} \text{ such that } \mathtt{A} \leq \mathtt{C} \text{ and } \mathtt{B} \leq \mathtt{C} & \\
\mathtt{A} \sqcup \mathtt{A} & = & \mathtt{A} & \text{(idempotent)} \\
\mathtt{A} \sqcup \mathtt{B} & = & \mathtt{B} \sqcup \mathtt{A} & \text{(commutative)} \\
\text{SELF\_TYPE}_{\mathtt{D}} \sqcup \mathtt{A} & = & \mathtt{D} \sqcup \mathtt{A} &
\end{array}$$

Let `T` and `F` be the static types of the branches of the conditional. Then the static type of the conditional is $\mathtt{T} \sqcup \mathtt{F}$. (think: *Walk towards* `Object` *from each of* `T` *and* `F` *until the paths meet.*)

NOTE: If either the second or third expression of a conditional has type `int` or `boolean`, then the other expression *must have the same type.*

7.6 Loops

A loop has the form

```
while <expr> loop <expr> pool
```

The predicate is evaluated before each iteration of the loop. If the predicate is `false`, the loop terminates and `null` is returned. If the predicate is `true`, the body of the loop is evaluated and the process repeats.

The predicate must have static type `boolean`. The body may have any static type. The static type of a loop expression is `Object`.

7.7 Blocks

A block has the form

```
{ <expr>; ... <expr>; }
```

The expressions are evaluated in left-to-right order. Every block has at least one expression; *the value of a block is the value of the last expression.* Each expression of a block may have any static type. The static type of a block is the static type of the last expression.

An occasional source of confusion in Wool is the use of semi-colons (";"). Semi-colons are used as terminators in lists of expressions (e.g., the block syntax above) and not as expression separators. Semi-colons also terminate other Wool constructs, see Section 11 for details.

7.8 Selection

A `select` expression has the form

```
        select
                <expr1> : <expr2>;
                ...
                <exprn-1> : <exprn>;
        end
```

A selection is a type of extended conditional. It consists of pairs of expressions where the first expression must have static type `boolean`.

Operationally, first expressions of each expression pair are evaluated. When one evaluates to `true`, the second expression of the pair is evaluated and that is the result of the selection. If no expressions evaluate to `true` then the result of the selection is the default initialization value of the selection type (see Section 5.1).

The static type of the selection is computed in a manner similar to conditionals (see Section 7.5). Apply the *join* operation to the second expressions of the pairs to determine the static type of the selection when the expressions are non-primitive. As with conditionals, if the static type of second expression of any pair is a primitive type then all second expressions must have the same static type, and this is the value of the selection.

7.9 new

A `new` expression has the form

```
new <type>
```

The value is a fresh object of the appropriate class. The static type is `<type>`. This may not be applied to primitive types.

7.10 isnull

The expression

```
isnull expr
```

evaluates to `true` if `expr` is `null` and evaluates to `false` if `expr` is not `null` .

7.11 Arithmetic and Comparison Operations

Wool has four binary arithmetic operations: `+, -, *, /`. The syntax is

```
expr1 <op> expr2
```

To evaluate such an expression first `expr1` is evaluated and then `expr2`. The result of the operation is the result of the expression. All of the binary arithmetic operations are left-associative.

Wool only supports integer arithmetic. The static types of the two sub-expressions must be `int`. The static type of the expression is `int`.

Wool has six comparison operations: `<, <=, =, ≅ >=, >`. These all have the same form as the binary operators:

```
        expr1 <op> expr2
```

The static type of these binary expressions is `boolean`. `<`, `<=`, `>`, and `>=` may only be applied to left and right expressions that have static type of `int`. Unlike arithmetic operators they do not chain. That is, `a < b > c` is an error; whereas `a + b - c` is not.

The comparisons `=` and `~=` are special cases. If either `<expr1>` or `<expr2>` has static type `int`, `boolean`, then the other must have the same static type. Any other types may be freely compared for equality. Equality operators associate to the right. This means that `a = b = c` is evaluated as `a = (b = c)`.

On non-basic objects (class instances), equality simply checks for pointer equality (i.e., whether the memory addresses of the objects are the same). If you want to have objects be comparable for equality and inequality using other than pointer equality, you need to define appropriate methods. Any non-basic object can be compared for equality to `null` .

Finally, there is one arithmetic and one logical unary operator. The expression `-<expr>` is the arithmetic negation of `<expr>`. The expression `<expr>` must have static type `integer` and the entire expression also has static type `integer`. The expression `~<expr>` is the boolean complement of `<expr>`. The expression `<expr>` must have static type `boolean` and the entire expression has static type `boolean`. Both of these unary operators associate to the right.

8 Basic Classes

In describing the basic classes we use the term `SELF_TYPE`. This refers to the dynamic type of the class in which the method is executed. For example, if class `C` inherits from `Object` directlyand `c` is an object of type `C`, then calling `c.copy()` returns an object of type `C`.

8.1 Object

The `Object` class is the root of the inheritance graph. Methods with the following declarations are defined:

```
abort() : Object
typeName() : Str
copy() : SELF_TYPE
```

The method `abort` halts program execution with an error message. The method `type_name` returns a string with the name of the class of the object. The method `copy` produces a *shallow* copy of the object.[4]

8.2 IO

The `IO` class provides the following methods for performing simple input and output operations:

```
outStr(x : Str) : SELF_TYPE
outInt(x : int) : SELF_TYPE
outBool(x : boolean) : SELF_TPE
inStr() : Str
inInt() : int
inBool() : boolean
```

[4]A shallow copy of a copies a itself, but does not recursively copy objects that a points to.

The methods `outStr`, `outInt` and `outBool` print their argument and return the object that executed the operation (ie., `this`). The method `in_string` reads a string from the standard input, up to but not including a newline character. The method `inInt` reads a single integer, which may be preceded by whitespace. Any characters following the integer, up to and including the next newline, are discarded by `inInt`. The method `inBool` reads a word that must be either `true` or `false`. The input is case-insensitive, so "TrUe" is valid. if the input is not `true` or `false` a runtime error occurs and the program aborts. Whitespace on either side of the input word is discarded.

A class can make use of the methods in the `IO` class by inheriting from `IO`. It is an error, however, to redefine the `IO` class.

8.3 Str

The `Str` class provides strings. The following methods are defined:

```
length() : int
concat(s : Str) : Str
substr(i : int, l : int) : Str
```

The method `length` returns the length of the string represented by `this`. The method `concat` returns the string formed by concatenating `s` after the string represented by `this`. The method `substr` returns the substring of its string represented by `this` beginning at position   with length `l`; string positions are numbered beginning at 0. A runtime error is generated if the specified substring is out of range.

The default initialization for variables of type `Str` is `""` (not `null`). It is an error to inherit from or redefine `Str`.

8.4 Primitives

There are two primitives that act as classes, `boolean` and `int`. These correspond to the Java primitives, `boolean` and `int`. Cool–W defines `Bool` and `Int` classes that cannot be extended and usually get implemented as the Java primitives. Wool avoids this by declaring the primitives as special types that cannot be extended (i.e. you may not create a a class that inherits either of these). The consequences of this decision is that the eventual implementation is simpler and operators like + just operate on the primitive objects.

9 Main Method

In the original Cool language, program must have a class `Main`. Furthermore, the `Main` class must have a method `main` that takes no formal parameters. Wool does not have this restriction. A source file that represents an executable program must contain all of the classes needed in the program (the Wool runner concatenates all of the files together when they are specified on the command line) and at least one of those classes must have a method called `main` that takes no arguments. This will be compiled as a static `main(String[])` Java method.

The remaining sections of this manual provide a more formal definition of Wool. Currently there are two sections covering lexical structure (Section 10) and grammar (Section 11).

10 Lexical Structure

The lexical units of Wool are numbers, type identifiers, object identifiers, special notation, strings, keywords, and white space.

10.1 Integers, Identifiers, and Special Notation

Numbers consist of integers. Integers are non-empty sequences of digits 0-9. Identifiers are sequences (other than keywords) consisting of letters, digits, and the underscore character. Type identifiers begin with a capital letter; object identifiers begin with a lower case letter. One other identifier, `this` is treated specially by Wool but is not treated as a keyword. The special syntactic symbols (e.g., parentheses, assignment operator, etc.) are given in Figure 1.

10.2 Strings

A string is a sequence of characters enclosed in double quotes `"..."`. The following are escape sequences that will be replaced by the appropriate characters.:

\t	tab
\n	newline
\\	backslash

A non-escaped newline character may not appear in a string:

```
"This \
is OK"
"This is not
OK"
```

In the first case, the string , as written in Java would be ”This \\\n is OK”. Any other character that is legal in a Java string may be included in a Wool string.

10.3 Comments

There are two forms of comments in Wool. Any characters between a pound sign “`#`” and the next newline (or EOF, if there is no next newline) are treated as comments. Comments may also be written by enclosing text in $(* \ldots *)$. This form of comment may be nested.

10.4 Keywords

The keywords of Wool are: **boolean, class, else, end, false, fi, if, int, inherits, isnull, loop, new, null, pool, select, then, true, while.** All keywords are case sensitive.

10.5 White Space

White space consists of any sequence of the characters: blank (ascii 32), `\n` (newline, ascii 10), `\t` (tab, ascii 9).

10.6 A small ambiguity

The original Cool language used the tilde character ('$\sim$') to represent the unary minus operator and the keyword *NOT* to represent logical negation. Wool overloads the `-` character to represent both the binary and unary arithmetic minus. Using '$\sim$' avoided any ambiguity when recognizing something like the assignment operator '`<-`' in an expression like `x<-1` and a comparison relation like `x<~1`. With the change for Wool, there is an ambiguity in the expression `x<-1`. We cannot determine lexically whether this expression consists of three tokens and assigns the integer 1 to the object `x` or if is a relation that is comparing the object `x` to the integer -1. Therefore we will require that when the two characters'`<-`' occur without any whitespace between them, then it represents the single token for the assignment operator. If the programmer wants to represent the relation, there must be at least one whitespace character between the '`<`' and '`-`' which represents two tokens for the less-than relation and the unary minus.

We actually do not allow spaces between the characters of any multi-character operator, but it is worth mentioning that we have introduced this ambiguity.

11 Wool Syntax

$$
\begin{array}{rcl}
\textit{program} & ::= & [\textit{class}]^{+} \\
\textit{class} & ::= & \textbf{class}\ \text{TYPE}\ [\textbf{inherits}\ \text{TYPE}]\ \{\ [\textit{vardef}\ \mid \textit{method}\,]^{*}\} \\
\textit{vardef} & ::= & \text{ID : TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ]^{?}; \\
\textit{method} & ::= & \text{ID(}\ [\ [\textit{formal}\ ]\ \ [, \textit{formal}\,]^{*}\ ]^{?}) : \text{TYPE}\{[\textit{vardef}\,]^{*}\,\textit{expr}\} \\
\textit{formal} & ::= & \text{ID : TYPE} \\
\textit{expr} & ::= & \text{ID}\ \texttt{<-}\ \textit{expr} \\
 & \mid & \textit{expr}.\text{ID(}\ [\ \textit{expr}\,[, \textit{expr}\,]^{*}\ ]^{?}\ ) \\
 & \mid & \text{ID(}\ [\ \textit{expr}\,[, \textit{expr}\,]^{*}\ ]^{?}\ ) \\
 & \mid & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{expr}\ \textbf{else}\ \textit{expr}\ \textbf{fi} \\
 & \mid & \textbf{while}\ \textit{expr}\ \textbf{loop}\ \textit{expr}\ \textbf{pool} \\
 & \mid & \{\ [\textit{expr}\ ;\ ]^{+}\} \\
 & \mid & \textbf{select}\ [\textit{expr} : \textit{expr};\ ]^{+}\ \textbf{end} \\
 & \mid & \textbf{new}\ \text{TYPE} \\
 & \mid & \textbf{isnull}\ \textit{expr} \\
 & \mid & \textit{expr} + \textit{expr} \\
 & \mid & \textit{expr} - \textit{expr} \\
 & \mid & \textit{expr} * \textit{expr} \\
 & \mid & \textit{expr}\,/\,\textit{expr} \\
 & \mid & \text{-}\ \textit{expr} \\
 & \mid & \textit{expr} < \textit{expr} \\
 & \mid & \textit{expr} <= \textit{expr} \\
 & \mid & \textit{expr} = \textit{expr} \\
 & \mid & \textit{expr}\ \sim\!=\!\textit{expr} \\
 & \mid & \textit{expr} >= \textit{expr} \\
 & \mid & \textit{expr} > \textit{expr} \\
 & \mid & \sim\ \textit{expr} \\
 & \mid & (\textit{expr}) \\
 & \mid & \text{ID} \\
 & \mid & \text{integer} \\
 & \mid & \text{string} \\
 & \mid & \textbf{true} \\
 & \mid & \textbf{false} \\
 & \mid & \textbf{null}
\end{array}
$$

Figure 1: Wool syntax.

Figure 1 provides a specification of Wool syntax. The specification is not in pure Backus-Naur Form (BNF); for convenience, we also use some regular expression notation. Specifically, A^* means zero or more A's in succession; A^+ means one or more A's, and $A^?$ means zero or one A's. Items in square brackets [...] are annotated with regular expression operators. These are optional items.

11.1 Precedence

The precedence of infix binary and prefix unary operations, from highest to lowest, is given by the following table:

```
.
- (unary)
isnull
* /
+ - (binary)
<=  <  >  >=
= ~=
~ (tilde, logical negation)
<-
```

Arithmetic and comparison operators have associativity as described in Section 7.11. [5]

12 Implementation Details

In this course you will implement a complete compiler. This means that you will generate code that runs on a particular machine implementation. We have chosen to use the Java JVM for several reasons. First, the JVM is widely used and is the machine that has been chosen for executing many languages (e.g. JRuby, Scala, Jython, etc.).

Code compiled from different source languages to the JVM can run in the same process, making it easy to have a program written in one language utilize libraries and routines compiled from another language. We do not support this in our implementation, but the extension to do this is quite easy.

This section provides guidelines for your implementation of Wool.

12.1 The Wool Namespace

The Java 9 release provides a `module` capability for grouping packages containing code, data, and resources. At some point we may adopt this for our code, but for now we use a simple approach to keeping our code in a single namespace.

All Wool classes will be compiled to be in a `wool` package. No sub-packages will be used. This is a very flat namespace and will work well for the course. The effect of this is that your generated classes should be compiled as if there were a Java "`package wool;`" line in the source code.

All of the basic classes described in Section 8 are implemented in the `wool` package. These classes are implemented in Java and are in the `woolLib.jar` library that must be in the runtime classpath. You will be given the source for the library so that you understand how it is implemented in order for you to use it with your code generator. If you find any errors in the library code, please report them to the instructor as soon as possible.

[5] It may be difficult to handle these associativity rules solely in the parser. They might be better validated in the semantic checking phases.

12.2 Source files

Typically a Wool program will contain all classes necessary for the program. However, the coolc tool (Section 12.3). However the Woolc tool will allow the compilation of many source files that are concatenated together to a single *virtual* file. Woolc generates one class file for each of the classes it compiles and places them in the appropriate directory.

12.3 The Woolc tool

In order to compile Wool source files, you need to have a tool for invoking the compiler and generating the appropriate output. You will be given code that implements a Woolc tool for my compiler. You can use it to fit it to your compiler.

The Woolc tool is a Java program and needs to be run as any other Java program. A typical command line invocation would be:

```
java -cp [classpath] {path to Woolc.class} [Woolc options] sourcefiles
```

You need to make sure that any runtime libraries or support classes are included in the classpath. It's a good idea to create a shell alias for running the Woolc tool. When I do this, I call the alias woolc.

12.3.1 woolc man page

```
NAME
    woolc - Wool compiler

SYNOPSIS
    woolc [ options ] [ sourcefiles ]

DESCRIPTION
    The woolc tool reads Wool source files containing Wool programs and
    compiles them into bytecode class files. Each Wool class is compiled
    to a separate bytecode class file. Source code file names must have .wl suffixes.

    All class files will be compiled and written to a "wool" directory that
    is a subdirectory of the current directory from where the compiler is
    executed (typically the directory returned from the pwd command).

    If there are multiple source files, the compiler reads them all and
    concatenates their contents for form one single text stream that is
    compiled. This makes it easier to separate classes that go together
    sometimes to be placed in separate files so that they can be reused.

OPTIONS
    -o directory
        This option sets the base destination directory for the output
        class files to the specified directory.

    -h help
        This option prints a synopsis of how to run the compiler.

    -p [phase] execute the compiler only through the specified phase
        Execute the compiler only through the phase indicated. The phases
        are:
        parse:    perform lexical and syntactic analysis only
        symbol:   perform any actions up to, but not including the
                  type-checking and final semantic analysis
        semantic: perform compilation through semantic analysis

    -d [internals] pretty-print the specified internal structures.
        These structures are:
            pt : the parse tree from ANTLR
            gui : show the parse tree using the ANTLR GUI
            tables : the symbol tables and any other tables that you
                use to perform type checking and semantic analysis
            source : the source code
```

12.4 Running Wool programs

You run Wool programs as you would any Java program. For example, assume that your current directory is called coolhome and have compiled your Wool class files into the coolhome/cool directory. One of your classes is called MyProgram and has a `main` method. You could run the program using the command:

```
java -cp .:pathToLibraries {program class}
```

13 Acknowledgements

Cool is based on Sather164, which is itself based on the language Sather. Portions of this document were cribbed from the Sather164 manual; in turn, portions of the Sather164 manual are based on Sather documentation written by Stephen M. Omohundro.

A number people have contributed to the design and implementation of Cool, including Manuel Fähndrich, David Gay, Douglas Hauge, Megan Jacoby, Tendo Kayiira, Carleton Miyamoto, and Michael Stoddart. Joe Darcy updated Cool to the version upon which Wool is based. Any errors in this manual are mine and should be reported to me, either in the Canvas project for your course or via email at gpollice@wpi.edu.