The Revised Report on the Woolite Programming Language

The programming assignments in this course will all be components of a single major project. The ultimate goal of this project will be to construct a complete compiler for the programming language Woolite. Woolite is a not very general purpose object-oriented programming language whose syntax derives largely from Java. In designing Woolite, I have attempted to eliminate as many unnecessary details as possible while including enough features to ensure that the construction of a Woolite compiler will expose you to many of the major issues one must address when constructing a compiler for a real language.

This document includes a formal description of the syntax of Woolite and an informal discussion of its semantics.

1 Lexical Issues

1.1 Keywords

The following strings are recognized as keywords of the Woolite language.

class	**else**	**extends**	**if**
int	**if**	**length**	**new**
null	**return**	**super**	**this**
void	**while**		

1.2 Identifiers

An identifier is a sequence of characters other than a keyword that begins with an alphabetic character and is composed of alphabetic characters and decimal digits.

1.3 Punctuation

The comma (‘,’) and semicolon (‘;’) are used as punctuation in contexts where the syntax of Woolite calls for a list of items. When used, commas separate list items while semicolons terminate items.

Parentheses (‘(’ and ‘)’) are used to delimit the formal and actual parameter lists associated with methods. Square brackets (‘[’ and ‘]’) are used around subscript expressions and array size specifiers. Braces (‘{’ and ‘}’) are placed before and after statement lists, the bodies of methods, and the bodies of classes.

1.4 Operators

The following symbols and pairs of symbols are recognized as operators of the Woolite language.

```
+   -   /   *
<   <=  >   >=
==  !=  !   &&
||
```

1.5 literals

An integer literal is an unbroken string of decimal digits.

A character literal is formed by placing a single, printable ASCII character or one of the escape sequences described below between a pair of single quotes.

Within character literals the following escape sequences can be used to specify that the special character shown should be used as the value of the literal. These escape sequences are necessary because tabs and newlines are not considered printable characters and therefore could not otherwise by used in character literals.

escape sequence	designated character
`\n`	newline
`\\`	backslash
`\'`	single quote
`\"`	double quote

A string literal consists of a sequence of printable ASCII characters and/or any of the escape sequences described above between a pair of double quotes.

1.6 White Space

Space characters and tabs are considered *white space* characters. Their presence is ignored except in character and strings literals and in cases where they terminate an identifier or integer literal. Newlines are also considered space characters with the further restriction that no lexical item can contain a newline character.

1.7 Comments

A pair of adjacent slashes ('//') indicates a comment. The slashes themselves and all characters following them on a line are ignored.

2 Program Structure

< program > → < class declaration >

Each program consists of a single class declaration (possibly containing other nested class declarations). This declaration must include the definition of a method named `main` that expects no parameters and returns a result of type `int`. When the program is run, execution will be initiated by constructing an instance of the class and invoking its `main` method. The value returned by the method will be displayed after the program terminates.

3 Class Declarations

< class declaration > → **class** < identifier > < superclass specification > {
< declaration list >
}

< superclass specification > → **extends** < identifier >
| ϵ

< declaration list > → < declaration list > < declaration >
| < declaration >

< declaration > → < class declaration >
| < method declaration >
| < variable declaration >

A class declaration is composed of a header specifying the class name and possibly the name of a superclass together with a class body which is simply a non-empty list of declarations of instance variables, methods and other classes. Each of the names declared within this list must be distinct from the other names in the list.

If a superclass name is included in the header of a class declaration that name must have been *previously* declared as a class. That is, superclass names are the one instance in which Woolite does not allow forward references to names declared later in the text of the program.

If a superclass name is provided, then the class defined by this declaration will be a subclass of the superclass specified. The subclass-superclass relationship is transitive. That is, if A is a superclass of B and B is a superclass of C, then A is a superclass of C.

A subclass inherits all method declarations included in any of its superclasses. Just as the names explicitly declared in a class must all be unique, the names declared in a class with superclasses must be distinct from the method names used in all its superclasses with the exception of method names that are overridden. A method declaration in a subclass overrides a method declaration in a superclass if the two declarations specify methods with the same name, return type, and parameter types.

4 Variable Declarations

< variable declaration > → < type specification > < identifier > ;

< type specification > → < type specification > [ ]
| < type name >

< type name > → < identifier >
| **int**

A variable declaration introduces a name which the program can associate with integer values, objects or arrays during execution depending on the type specified in the variable's declaration. Variable declarations may appear in the body of a class declaration or method declaration. The scope of a variable declaration is the class or method body within which it occurs. Variable's, including the instance variables of a class, cannot be accessed outside this scope.

The type specification used in a variable declaration consists of either a class name or the keyword **int**, followed by a possibly empty list of pairs of square brackets. If no square brackets are present, the name will be used to refer to a single value or object of the specified type name. If n pairs of brackets are included, then the name will be used to refer to an n-dimensional array of the type specified. These forms of type specifications are also used in method parameter declarations, method return type specifications and in local variable declarations.

5 Methods

< method declaration > → < return type > < identifier > < formals part > {
< method body >
}

< return type > → **void**
| < type specification >

< formals part > → (< formals list >)
| ( )

< formals list > → < formals list >, < formal declaration>
| <formal declaration>

< formal declaration > → < type specification > < identifier >

< method body > → < variable declaration list > < statement list >

< variable declaration list > → < variable declaration list > < variable declaration >
| ϵ

Method declarations consist of a method header specifying the method's name, return type and parameter specifications followed by a method body which may contain local variable declarations and must contain a list of statements to execute when the method is invoked.

The name of a method must be distinct from all other names declared within its class. It must also be distinct from the method names declare in any of the superclasses of its class unless the method's name was previously used to define a method with the same name, return type and parameter types. In this case, the new method declaration overrides the declaration found in the superclass.

The return type specified in a method's declaration must either be `void` or of the same form used to specify the type of a variable declaration. If the return type is `void`, then invoking the method will not produce any value. This implies that the method's body need not specify a return value and that invocations of the method may only be used as statements, not as expressions. If the return type is not void, then any `return` statement included within the method's body must include an expression describing a value or object of this type or of a class which is a subclass of the return type. Note that this implies that no subtyping is allowed when returning an array value. Woolite methods are not required to include a `return` statement. If a non-void method terminates without executing a `return`, a default value will be returned (0 or `null`).

The formal parameter declarations included in a method header describe the number and types of parameters required when the method is invoked. They also introduce names that can be used to refer to the actual parameter values provided when the method is invoked. For the purposes of scoping, formal declarations are treated as if they occurred within the body of the method. This implies that the formal parameter names of a method must be distinct from the names of variables declared within its body.

The scope of the name associated with a method declaration is the body of the class in which the declaration occurs. In particular, there is no restriction against forward references to methods. A method name may also be accessed outside of this scope when used in a method invocation of the form

`o.m(...)` where `o` is an expression that refers to an object of the class in which the method `m` is declared. This is true for all methods. That is, all methods are implicitly public.

Methods may be called recursively.

The body of a method is composed of a possible empty list of local variable declarations followed by the statements to be executed when the method is invoked. Note that Woolite does not allow nesting of class or method declarations within a method declaration.

6 Statements

< statement list > → < statement list > < statement >
| < statement >

< statement > → < invocation > ;
| < assignment statement> ;
| < return statement > ;
| < if statement >
| < while statement >
| < for statement >

6.1 Invocations

< invocation > → < identifier > < actuals >
| < primary factor > . < identifier > < actuals >
| **super** . < identifier > < actuals >

< actuals > → (< actual list >)
| ( )

< actual list > → < actual list > , < expression >
| < expression >

Syntactically, a method invocation consists of a possibly qualified method name followed by a list of expressions describing the actual parameter values to be passed to the method.

There are three ways to identify the method to be invoked. One may specify an expression that describes an object followed by a period and the name of a method. In this case, the type associated with the expression must be a class that includes a method with the specified name. The method actually invoked, however, may be different from the method declared in this class, however, because all methods are virtual. If the actual object produced by the expression is a member of a subclass of the type associated with the expression at compile time, and this subclass has overridden the method name, then the definition of the method associated with the subclass will be invoked.

If an unqualified name is used, then the method invoked is determined using standard static scope rules. That is, whenever a method is invoked, the invocation is associated with some object belonging to the class in which the method was defined (or one of its subclasses). We will call this object the active object. In the case of a qualified invocation, the active object is just the object described by the qualifying expression. If the method name used in an unqualified invocation is defined within the class in which it is referenced, then the active object for the new invocation will be the same as for the current invocation. If the method name is declared by a class in which the class that contains the

invocation is itself nested, then the new active object will be determined as it would be if the invocation has been encountered in the same context in which the construction that created the current active object occurred.

All method invocations, however, are virtual. Therefore, if a class contains declarations of two methods m_1 and m_2 and an invocation of the form $m_1(...)$ appears in the body of m_2, it is not necessarily the case that the method body executed is the body associated with m_1 in the class declaration. If the object belongs to a subclass in which m_1 has been overridden, the version of m_1 defined in the subclass will be used.

Finally, if **super** is used, then the method invoked is still assumed to be associated with the same object as the method that contains the invocation, but the definition used is that associated with the immediate superclass of the object's class even if the method was overridden in the object's class.

The number of actual parameter expressions included in an invocation must match the number of formal parameters declared in the method's definition. The type of each actual parameter expression must match the type of the corresponding formal or be a subclass of the class used to specify the formal parameter's type. Note that this implies that no subtyping is allowed when passing an array value as a parameter.

A method invocation is executed by evaluating the actual parameter specifications (in an unspecified order), associating the values obtained with the formal parameter names and then executing the statements found in the method declaration's body.

Invocations can appear as statements and as subparts of expressions. Method's declared to return void cannot be used in invocations used as expressions.

6.2 Assignment

< assignment statement > → < variable > = < expressions >

The value produced by evaluating the expression provided on the right side of the equal sign is associated with the variable found on the right side. The type specified in the variable's declaration and the type of the value produced by the expression must either be identical or the type of the value produced by the expression must be a class which is a subtype of the class specified as the variable's type. Note that this implies that no subtyping is allowed when assigning a name to an array value.

6.3 Return Statements

< return statement > → **return**
| **return** < expression >

A return statement causes return of control to the invoker of the method in which it is executed. Executing a return statement in the main method of the program's outer class causes program termination.

Return statements placed in methods declared to return **void** may not include an expression. Return statements placed in all other methods must include an expression. The type of this expression must either be identical to the return type or a subclass of the return type. This expression is evaluated and its value returned as the value of the method invocation.

There is an implicit return statement at the end of the body of each method. If the method's return type is **int**, this implicit return will return 0. If the return type is a class type or array type, it will return null.

6.4 If Statements

< if statement > → **if** (< expression >)< statement part >
< else part >

< else part > → **else** < statement part >
| ϵ

< statement part > → { < statement list > }
| < statement >

Woolite does not recognize Boolean values as a separate type. Instead, it treats the integer value 0 as 'false' and any non-zero value as 'true'. Thus, the expression used as a condition in an if statement must have type **int**.

To execute an if statement, the expression is first evaluated. If its value is non-zero, the statement or statement list following the expression is executed after which execution resumes with the statement following the if statement. If the expression's value is zero and a non-empty else part was included, the statement or statement list found in the else part is executed. If the else part is empty and the expression's value is zero, then control simply passes to the statement after the if statement.

6.5 While Statements

< while statement > → **while** (< expression >)
< statement part >

The execution of a while loop begins with the evaluation of the expression included in the statement's header. This expression's type must be **int**. If this expression's value is zero, execution of the while loop terminates immediately. Otherwise, the statement or statement list included is executed repeatedly until the value of the expression becomes zero.

6.6 For Statements

< for statement > → **for** (< assignment statement > ;
< expression > ;
< assignment statement >)
< statement part >

The syntax of the **for** statement in Woolite is based on the most common use of the corresponding statement in C and Java rather than on the actual syntax of the C and Java for statement. In particular, the first and third items in the header of the for can only be assignment statements.

The first step in the execution of a for statement is the execution of the first assignment in its header. After this is done, the expression included in the header is evaluated. This expression must have type `int`. If its value is 0, the execution of the statement is complete. Otherwise, the statement part and the assignment included as the third component of the header are executed repeatedly until evaluation of the expression yields 0.

7 Variables

< variable > → < identifier >
| < primary factor > [< expression >]

A variable is either a simple identifier or an expression that produces an array value followed by an array subscript expression. If a variable is a simple identifier the identifier must be bound by a variable declaration or a formal parameter declaration in an enclosing scope. If the variable is an expression followed by a subscript expression, the sub-variable must denote an array and the value of the expression at run-time must be non-negative and less than the number of elements in the array.

Note that Woolite does not allow one to form a variable by placing a dot between an expression and an identifier as in `o.v`. Therefore, it is not possible to access an instance variable of an object from outside the object. All instance variables in Woolite are effectively private.

8 Expressions

8.1 Logical Expressions

< expression > → < logical term >
| < expression > `||` < logical term >

< logical term > → < logical factor >
| < logical term > `&&` < logical factor >

< logical factor > → < relational expression >
| `!` <relational expression>

Woolite provides the logical 'and' (`&&`), 'or' (`||`) and 'not' (`!`) operators. In Woolite, these operators are actually applied to integers. They interpret their operands using the usual scheme: 0 represents 'false' and any non-zero value represents 'true'. The type of a logical expression in Woolite is **int**. The logical and relational operators of Woolite all produce the value 1 when the result of an operation is 'true'.

The `!` operator has the highest precedence among the logical operators followed by `&&` and finally `||`. Operators of equal precedence are grouped from left to right.

The `&&` and `||` operators are evaluated lazily in Woolite. That is, if the first operand of an `&&` operator is zero, a result of zero is returned without ever evaluating the second operand. Similarly, if the first operand of an `||` operator is non-zero, the second operand will not be evaluated.

8.2 Relational Expressions

< relational expression > → < arithmetic expr. >
| < arithmetic expr. > < relational > < arithmetic expr. >

<relational> → `<` |`>` |`=` |`<=` |`>=` |`!=`

Relational expressions can be used to compare the values produced by two expressions. Note that the syntax of the language prohibits the expression

```
a < b < c
```

although it allows the essentially equivalent expression

```
(a < b) < c
```

The type of all relational expressions is **int**. The operands of all relational operators except == must be `int`. The == operator can be applied to two expressions of any type as long as one of the two types is equivalent to or a subtype of the other.

8.3 Arithmetic expressions and primary expressions

< arithmetic expr. > → < term >
| < arithmetic expr. > < adding operator > < term >

< adding operator > → `+` | `-`

< term > → < factor >
| < term > < multiplying operator > < factor >

< multiplying operator > → `*` | `/`

< factor > → < adding operator > < factor >
| **this**
| **null**
| < literal >
| < expression > . **length**
| < primary factor >
| < construction >

< primary factor > → (< expression >)
| < variable >
| < invocation >
| < type cast >

< type cast > → (< ident >) < primary factor >

Woolite provides the usual addition, subtraction and multiplication operations on integers. All these operators produce expressions of type **int**.

The unary '`+`' and '`-`' have the highest precedence among arithmetic operators. Multiplying operators have the next highest precedence followed by adding operators. Operators of equal precedence group from left to right.

The keyword **this** may be used as an expression to denote the current object. This expression's type will be the name of the immediately enclosing class declaration. Note that because **this** is classified as a factor rather than a primary factor it is not possible to write a method invocation of the form `this.m(...)`.

The keyword **null** may be used as an expression to denote the value associated with a class or array variable that does not currently refer to any object. This expression has a type that is treated as a subtype of all class and array types.

Following an expression of an array type with a period and the keyword **length** produces an integer valued expression that will return the number of elements in the array value described by the expression.

A type cast is formed by preceding a primary factor with a parenthesized class name. The static type of such an expression is the class named. The static type of the primary factor must be a superclass of this type name. At execution time, the actual type of the value produced by the primary factor will be compared to the class name specified in the cast. If the actual type of the object is a subtype of the specified class, then the object produced by the primary factor will be the result of the cast. Otherwise, the value returned by the cast will be **null**.

8.4 literals

< literal > → < integer literal >
| < string literal >
| < character literal >

Both integer literals and character literals produce values of type integer. The value of a given character literal is the integer value of the number used to represent the character in the ASCII code. Evaluation of a string literal creates a new integer array whose length is equal to the length of the string and whose elements are initialized to the ASCII codes for the characters in the string.

8.5 Invocations

An invocation may be used as an expression as long as the return type associated with the method invoked is not void. The type of such an expression is the return type of the method.

8.6 Constructions

< construction > → **new** < type descriptor > ( )

< type descriptor > → < identifier >
| < array descriptor >

< array descriptor > → < type name > [< expression >]
| < array descriptor > [ ]

A construction is used to produce a new object or array in Woolite. If an object is being constructed, the type descriptor included must be a class name. The construction is executed by allocating memory for all the instance variables declared in the class specified and all of its superclasses. These variables are then initialized so that all integers are 0 and all variables with class types and array types are null. The type of such a construction is the class of the object created.

If an array is being constructed the type specifier must consist of either a class name or **int** followed by an integer expression in square brackets followed by 0 or more pairs of empty square brackets. The integer expression determines the number of elements in the array. If n pairs of square brackets are provided (including the ones surrounding the expression), the type associated with the construction will be an n dimensional array of the base type specified. In Woolite, n dimensional arrays are implemented as 1 dimensional arrays of references to n-1 dimensional arrays. Therefore, when an array is constructed, its elements are initialized to null unless it is a one dimensional array of integers in which case the elements are initialized to 0.

9 Input/Output

In the proud tradition of Algol 60, the definition of Woolite includes no mechanisms for input or output. It is assumed that any implementation of the language will include appropriate built-in methods to provide the needed facilities.