UnCoolAid: UnCool Reference Manual

1 Introduction

This manual describes the programming language UnCool, which is based on the language Cool - the Classroom Object-Oriented Language - developed by Alex Aiken It is quite close to Cool, but with a simpler type system, one or two of its more difficult features have been omitted, and it has been extended to include simple arrays. The syntax is almost identical to Cool and it will compile and run many "real" Cool programs. In the remainder of this manual, all references to Cool in fact refer to the language UnCool.

UnCool programs are sets of classes. A class encapsulates the variables and procedures of a data type. Instances of a class are objects. In UnCool, 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. There is no inheritance, so the language is really object-based, rather than object-oriented. In addition to classes, UnCool has five *primitive types* : Int, String, Bool, Float, and Int[] (single dimensional array of Int).

UnCool is an expression language. Most UnCool constructs are expressions, and every expression has a value and a type. UnCool 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 UnCool, it guarantees that no runtime type errors can arise in the execution of UnCool programs.

This document starts with an informal description of the features. The syntax and lexical information is at the end of the document.

NOTE: We will not be implementing all of the features of this language. See lab specifications to determine what is required.

2 Classes

All code in Cool is organized into classes. All class names are globally visible. Class names begin with an uppercase letter. Classes may not be redefined. Class definitions have the form:

```
class <classname> {
    <feature_list>
};
```

2.1 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 objects of class A. A method of class A is a procedure that may manipulate the variables and objects 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. Cool supports information hiding through a simple mechanism: all attributes have scope local to the class, and all methods have global scope. Thus, the only way to provide access to object state in Cool 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, but a method and an attribute may have the same name.

The following code fragment illustrates simple cases of both attributes and methods:

```
class Stack {
      stack : Int[];
     top : Int;

      push(item: Int) : Int { ... };
     pop() : Int { ... };
     init(size: Int): Stack {stack <- new Int[size]; top <- 0; self};
    ...
};
```

In this example, the class `Stack` has two attributes (`stack` and `top`), two methods (`push()` and `pop()`) and a constructor `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. ‘self’ is a keyword that indicates the current class.

Given object s of class `Stack`, we can modify the given attributes as follows:

```
s.push(2);
```

This notation is *object-oriented dispatch*. There may be many definitions of `push()` methods in many different classes. The dispatch looks up the class of the object `s` to decide which `push`() method to invoke. Because the class of `s` is `Stack`, the `push()` method in the `Stack` class is invoked. Within the invocation, the variables refer to s's attributes.

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.

```
s <- new Stack(10);
```

This will cause the ‘init’ method to execute with the given parameter. If no init function is defined for a given class, then any attributes get the default initialization at creation time. There is no mechanism in UnCool (or Cool) for programmers to deallocate objects. "Real" Cool has *automatic memory management*; objects that cannot be used by the program are deallocated by a runtime garbage collector - in UnCool we have left out the garbage collector, which can lead to memory leaks and heap overflows!

2.1.1 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.

All variables in Cool are initialized to contain values of the appropriate type. The special value `void` is a member of all non-primitive types (and a member of the type Int[]) and is used as the default initialization for variables of these types where no initialization is supplied by the user. Note that there is no name for `void` in Cool; the only way to create a `void` value is to declare a variable of some type other than Int, String, or Bool and allow the default initialization to occur.

There is a special form `isvoid` expr that tests whether a value is `void`. In addition, `void` values may be tested for equality. A `void` value may be passed as an argument, assigned to a variable, or otherwise used in any context where any value is legitimate, except that a dispatch on `void` generates a runtime error.

2.1.2 Methods

A method definition has the form

```
<id>(<id> : <type>,...,<id> : <type>): <type> { <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.

2.2 Main Class

Every program must have a class Main. Furthermore, the Main class must have a method main that takes no formal parameters and returns an Int. A program is executed by evaluating (new Main).main().

3 Expressions

Expressions are the largest syntactic category in UnCool

3.1 Constants

The simplest expressions are constants. The Bool constants are `true` and `false`. Int constants are unsigned strings of decimal digits such as 0, 123, and 007. String constants are sequences of characters enclosed in double quotes, such as "This is a string." String constants may be at most 1024 characters long. There are no special or control characters allowed in string constants.

3.2 Identifiers

The names of local variables, formal parameters of methods, and class attributes are all expressions. Local variables and formal parameters have lexical scope. Attributes are visible throughout a class in which they are declared, 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.

3.3 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>.

3.4 Dispatch

There are two forms of dispatch in UnCool. The two forms differ only in how the called method is selected. The most commonly used form of dispatch is

<id1>.<id2>(<expr>,...,<expr>)

Consider the dispatch x.f(e_1,...,e_n). To evaluate this expression, the arguments are evaluated in left-to-right order, from e_1 to e_n. Next, id1 is evaluated and its class C noted. Finally, the method id2 in class C is invoked, the actual arguments are bound to the formals as usual. The value of the expression is the value returned by the method invocation.

The other form of dispatch is:

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

This form is shorthand for self.<id>(<expr>,...,<expr>).

3.5 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 Bool. The branches may have any static types. However, the two branches must have the same static type.

3.6 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 void is returned. If the predicate is true, the body of the loop is evaluated and the process repeats.

The predicate must have static type Bool. The body may have any static type. The static type of a loop expression always Int and the returned value is always 0.

3.7 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. The expressions of a block may have any static types. The static type of a block is the static type of the last expression.

An occasional source of confusion in Cool 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 Cool constructs, see grammar for details.

3.8 Let

A let expression has the form

```
let
<id1> : <type1> [ <- <expr1> ], ..., <idn> : <typen> [ <- <exprn> ]
in <expr>
```

The optional expressions are *initialization*; the other expression is the *body*. A let is evaluated as follows. First <expr1> is evaluated and the result bound to <id1>. Then <expr2> is evaluated and the result bound to <id2>, and so on, until all of the variables in the let are initialized. (If the initialization of <idk> is omitted, the default initialization of type <typek> is used.) Next the body of the let is evaluated. The value of the let is the value of the body.

The let identifiers <id1>,...,<idn> are visible in the body of the let. Furthermore, identifiers <id1>,...,<idk> are visible in the initialization of <idm> for any m > k.

If an identifier is defined multiple times in a let, later bindings hide earlier ones. Identifiers introduced by let also hide any definitions for the same names in containing scopes.
Every let expression must introduce at least one identifier.

The type of an initialization expression must conform to the declared type of the identifier. The type of let is the type of the body.

The scope of a let extends as far (encompasses as many tokens) as possible.

3.9 New and New Int[]

A new expression has the form

```
new <type>
```

The value is a fresh object of the appropriate class. The static type is <type>. The other form of new is

```
new Int '[' expr ']'
```

where expr is an expression that evaluates to an Int. The static type is Int[]. Note that variables of type Int[] are *references* (i.e. pointers) to an array that must be allocated as above from the heap (this is similar to Java.)

3.10 Isvoid

The expression

```
isvoid expr
```

evaluates to true if expr is void and evaluates to false if expr is not void.

3.11 Arithmetic and Comparison Operations

UnCool has three 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. The static types of the two sub-expressions must be Int. The static type of the expression is Int.

UnCool has three comparison operations: <, <=, =. For < and <= the rules are exactly the same as for the binary arithmetic operations, except that the result is a Bool. The comparison is a special case. If either <expr1> or <expr2> has static type Int, Bool, or String, then the other must have the same static type. Any other types may be freely compared. On non-basic objects, equality simply checks for pointer equality (i.e., whether the memory addresses of the objects are the same). Equality is defined for void.

In principle, there is nothing wrong with permitting equality tests between, for example, Bool and Int. However, such a test must always be false and almost certainly indicates some sort of programming error. The UnCool type checking rules catch such errors at compile-time instead of waiting until runtime.

Finally, there is one arithmetic and one logical unary operator. The expression ~ <expr> is the integer complement of <expr>. The expression <expr> must have static type Int and the entire expression has static type Int. The expression not <expr> is the boolean complement of <expr>. The expression <expr> must have static type Bool and the entire expression has static type Bool.

7.12 Array Reference

The notation

```
<id> '[' expr ']'
```

whether on the left or right hand side on an assignment refers to an element of the integer array referenced by <id>, indexed by the expression <expr>. The index value should be between 0 (zero) and one less than the size of the array, as allocated via new Int[]. The type of the resulting value is Int. The variable <id> must be of type Int[]. It is an error to attempt to reference an array that has not been properly allocated (i.e. if <id> is void).

4 Input-Output

UnCool provides the following built-in pseudo-methods for performing simple input and output operations:

```
out_string(x : String) : Int
out_int(x : Int) : Int
in_int() : Int
```

The methods `out_string` and `out_int` print their argument and return an error code. The method `in_int` 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 `in_int`. These are not "normal" methods, are not invoked on an Object, so syntactically, they are simply expressions.

5 Base Types

5.1 Int

The Int type provides integers. Int is not an Object, so there are no methods special to Int. The default initialization for variables of type Int is 0 (not void).

5.2 String

The String type provides printable strings. A String is not a true Object and there are no methods defined for Strings. A String variable is a reference to an actual String instance, in the same manner as a "char *" in the language C. A String can be assigned to a variable of type String, passed as an argument to a method, or as an argument to `out_string()`. There are no operations defined for Strings, and the only really useful thing you can do with them is to print them. The default initialization for variables of type String is "" (not void).

5.3 Bool

The Bool type provides `true` and `false`. The default initialization for variables of type Bool is false (not void).

5.4 Int[]

The Int[] type provides a simple one-dimensional arrays of Int. An Int variable holds a reference to such an array, which must first be allocated from the heap with a call to new Int[expr]. Assigning a variable of type Int[] simply assigns the reference (i.e. the pointer to the array): it does not copy or alter the contents of the array in any way. In this sense, think of it as an int* pointer in "C". Array elements are numbered from 0 (zero).

6 Lexical Structure

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

6.1 Integers, Identifiers, and Special Notation

Integers are non-empty strings of digits 0-9. Identifiers are strings (other than keywords) consisting of letters, digits, and the underscore character. Type identifiers begin with a capital letter; method names, and attribute and let variable identifiers begin with a lower case letter. The special syntactic symbols (e.g., parentheses, assignment operator, etc.) are

( ) [ ] { } , ; : <- + - * < <= = ~

6.2 Strings

Strings are enclosed in double quotes "...". Within a string, a sequence `\c' denotes the character `c', with the exception of `\n' which denotes a newline.

6.3 Comments

There is one form of comment in UnCool. Any characters between two dashes " - -" and the next newline (or EOF, if there is no next newline) are treated as comments.

6.4 Keywords

The keywords of UnCool are: **Bool**, **class, else, false, fi, if, in, Int, isvoid, let, loop, pool, self, String, tel, then, while, new, of, not, true.** Keywords are case sensitive.

6.5 White Space

White space consists of any sequence of the characters: blank, newline, \f, \r, \t, \v.

7 UnCool Grammar