CoolAid: The Cool Reference Manual*

Contents

1 Introduction

This manual describes the programming language Cool: the *C*lassroom *O*bject-*O*riented *L*anguage. Cool is a small language that can be implemented with reasonable effort in a one semester course. Still, Cool retains many of the features of modern programming languages including objects, static typing, and automatic memory management.

Cool programs are sets of *classes*. A class encapsulates the variables and procedures of a data type. Instances of a class are *objects*. In Cool, 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.

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

This manual is divided into informal and formal components. For a short, informal overview, the first half (through Section 9) suffices. The formal description begins with Section 10.

2 Getting Started

The reader who wants to get a sense for Cool at the outset should begin by reading and running the example programs in the directory `~cs164/examples`. Cool source files have extension `.cl` and Cool assembly files have extension `.s`. The Cool compiler is `~cs164/bin/coolc`. To compile a program:

```
coolc [ -o fileout ] file1.cl file2.cl ... filen.cl
```

The compiler compiles the files `file1.cl` through `filen.cl` as if they were concatenated together. Each file must define a set of complete classes—class definitions may not be split across files. The `-o` option specifies an optional name to use for the output assembly code. If `fileout` is not supplied, the output assembly is named `file1.s`.

The `coolc` compiler generates MIPS assembly code. Because not all of the machines the course is using are MIPS-based, Cool programs are run on a MIPS simulator called `spim`. To run a cool program, type

```
% spim
(spim) load "file.s"
(spim) run
```

To run a different program during the same `spim` session, it is necessary to reinitialize the state of the simulator before loading the new assembly file:

```
(spim) reinit
```

An alternative—and faster—way to invoke `spim` is with a file:

```
spim -file file.s
```

This form loads the file, runs the program, and exits `spim` when the program terminates. Be sure that `spim` is invoked using the script `~cs164/bin/spim`. There may be another version of `spim` installed in on some systems, but it will not execute Cool programs. An easy way to be sure of getting the correct version is to alias spim to ~cs164/bin/spim. The spim manual is available on the course Web page and in the course reader.

The following is a complete transcript of the compilation and execution of ~cs164/examples/list.cl. This program is very silly, but it does serve to illustrate many of the features of Cool.

```
% coolc list.cl
% spim -file list.s
SPIM Version 5.6 of January 18, 1995
Copyright 1990-1994 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README a full copyright notice.
Loaded: /home/ee/cs164/lib/trap.handler
5 4 3 2 1
4 3 2 1
3 2 1
2 1
1
COOL program successfully executed
%
```

3 Classes

All code in Cool 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>
};
```

The notation [...] denotes an optional construct. All class names are globally visible. Class names begin with an uppercase letter. Classes may not be redefined.

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 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.

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

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

isNil() : Bool { false };

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

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 method `init`:

```
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 `self` 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 for programmers to deallocate objects. Cool has *automatic memory management*; objects that cannot be used by the program are deallocated by a runtime garbage collector.

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 { ... };
```

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

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 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 has four other *basic classes*: `Int`, `String`, `Bool`, and `IO`. The basic classes are discussed in Section 8.

4 Types

In Cool, every class name is also a type. In addition, there is a type `SELF_TYPE` that can be used in special circumstances.

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, whether that is in a `let`, `case`, or as the formal parameter of a method. The types of all attributes must also be declared.

The basic type rule in Cool 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). As discussed above, 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 $\mathtt{C} \leq \mathtt{P}$
- 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 SELF_TYPE

The type `SELF_TYPE` is used to refer to the type of the `self` variable. This is useful in classes that will be inherited by other classes, because it allows the programmer to avoid specifying a fixed final type at the time the class is written. For example, the program

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

```
class Silly {
   copy() : SELF_TYPE { self };
};

class Sally inherits Silly { };

class Main {
   x : Sally <- (new Sally).copy();

   main() : Sally { x };
};
```

Because SELF_TYPE is used in the definition of the copy method, we know that the result of copy is the same as the type of the self parameter. Thus, it follows that (new Sally).copy() has type Sally, which conforms to the declaration of attribute x.

Note that the meaning of SELF_TYPE is not fixed, but depends on the class in which it is used. In general, SELF_TYPE may refer to the class C in which it appears, or any class that conforms to C. When it is useful to make explicit what SELF_TYPE may refer to, we use the name of the class C in which SELF_TYPE appears as an index $\texttt{SELF\_TYPE}_{\texttt{C}}$. This subscript notation is not part of Cool syntax—it is used merely to make clear in what class a particular occurrence of SELF_TYPE appears.

From Definition 4.1, it follows that $\texttt{SELF\_TYPE}_{\texttt{X}} \leq \texttt{SELF\_TYPE}_{\texttt{X}}$. There is also a special conformance rule for SELF_TYPE:

$$\texttt{SELF\_TYPE}_{\texttt{C}} \leq \texttt{P} \text{ if } \texttt{C} \leq \texttt{P}$$

Finally, SELF_TYPE may be used in the following places: new SELF_TYPE, as the return type of a method, as the declared type of a let variable, or as the declared type of an attribute. No other uses of SELF_TYPE are permitted.

4.2 Type Checking

The Cool type system guarantees at compile time that execution of a program cannot result in runtime 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 $\texttt{D}_{\texttt{e}}$ be a dynamic type of e and let $\texttt{S}_{\texttt{e}}$ 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 $\texttt{D}_{\texttt{e}} \leq \texttt{S}_{\texttt{e}}$.

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 runtime 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. 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 Void

All variables in Cool are initialized to contain values of the appropriate type. The special value `void` is a member of all types and is used as the default initialization for variables where no initialization is supplied by the user. (`void` is used where one would use `NULL` in C or `null` in Java; Cool does not have anything equivalent to C's or Java's `void` type.) 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 class other than `Int`, `String`, or `Bool` and allow the default initialization to occur, or to store the result of a `while` loop.

There is a special form `isvoid expr` that tests whether a value is `void` (see Section 7.11). 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 to or case on `void` generates a runtime error.

Variables of the basic classes `Int`, `Bool`, and `String` are initialized specially; see Section 8.

6 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.

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(): String { "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 Cool.

7.1 Constants

The simplest expressions are constants. The boolean constants are `true` and `false`. Integer constants are unsigned strings of 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 other restrictions on strings; see Section 10.

The constants belong to the basic classes `Bool`, `Int`, and `String`. The value of a constant is an object of the appropriate basic class.

7.2 Identifiers

The names of local variables, formal parameters of methods, `self`, and class attributes are all expressions. The identifier `self` may be referenced, but it is an error to assign to `self` or to bind `self` in a `let`, a `case`, or as a formal parameter. It is also illegal to have attributes named `self`.

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 `self`, 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 three forms of dispatch (i.e. method call) in Cool. The three 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 void a runtime error is generated). Finally, the method f in class C is invoked, with the value of e_0 bound to self 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 runtime, 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. Otherwise, if f has return type SELF_TYPE, then the static type of the dispatch is A. To see why this is sound, note that the self parameter of the method f conforms to type A. Therefore, because f returns SELF_TYPE, we can infer that the result must also conform to A. Inferring accurate static types for dispatch expressions is what justifies including SELF_TYPE in the Cool type system.

The other forms of dispatch are:

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

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

The second form provides a way of accessing methods of parent classes that have been hidden by redefinitions in child classes. Instead of using the class of the leftmost expression to determine the method, the method of the class explicitly specified is used. For example, e@B.f() invokes the method f in class B on the object that is the value of e. For this form of dispatch, the static type to the left of "@" must conform to the type specified to the right of "@".

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 Bool. 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 types other than SELF_TYPE. The *least type* of a set of types means the least element with respect to the conformance relation $\leq$.

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

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

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 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 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. 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 Section 11 for details.

7.8 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 <expr> of a let extends as far (encompasses as many tokens) as the grammar allows.

7.9 Case

A case expression has the form

```
case <expr0> of
    <id1> : <type1> => <expr1>;
    . . .
    <idn> : <typen> => <exprn>;
esac
```

Case expressions provide runtime type tests on objects. First, expr0 is evaluated and its dynamic type C noted (if expr0 evaluates to void a run-time error is produced). Next, from among the branches the branch with the least type <typek> such that $C \leq$ <typek> is selected. The identifier <idk> is bound to the value of <expr0> and the expression <exprk> is evaluated. The result of the case is the value of <exprk>. If no branch can be selected for evaluation, a run-time error is generated. Every case expression must have at least one branch.

For each branch, let T_i be the static type of <expri>. The static type of a case expression is $\bigsqcup_{1 \leq i \leq n} T_i$. The identifier id introduced by a branch of a case hides any variable or attribute definition for id visible in the containing scope.

The case expression has no special construct for a "default" or "otherwise" branch. The same affect is achieved by including a branch

```
x : Object => ...
```

because every type is $\leq$ to Object.

The case expression provides programmers a way to insert explicit runtime type checks in situations where static types inferred by the type checker are too conservative. A typical situation is that a programmer writes an expression e and type checking infers that e has static type P. However, the programmer may know that, in fact, the dynamic type of e is always C for some $C \leq P$. This information can be captured using a case expression:

```
case e of x : C => ...
```

In the branch the variable x is bound to the value of e but has the more specific static type C.

7.10 New

A new expression has the form

```
new <type>
```

The value is a fresh object of the appropriate class. If the type is SELF_TYPE, then the value is a fresh object of the class of self in the current scope. The static type is <type>.

7.11 Isvoid

The expression

```
isvoid expr
```

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

7.12 Arithmetic and Comparison Operations

Cool 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.

The static types of the two sub-expressions must be Int. The static type of the expression is Int. Cool has only integer division.

Cool 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, including SELF_TYPE, 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 Cool 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.

8 Basic Classes

8.1 Object

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

```
abort() : Object
type_name() : String
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.[3]

8.2 IO

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

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

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

The methods `out_string` and `out_int` print their argument and return their `self` parameter. The method `in_string` reads a string from the standard input, up to but not including a newline character. 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`.

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

8.3 Int

The `Int` class provides integers. There are no methods special to `Int`. The default initialization for variables of type `Int` is 0 (not `void`). It is an error to inherit from or redefine `Int`.

8.4 String

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

```
length() : Int
concat(s : String) : String
substr(i : Int, l : Int) : String
```

The method `length` returns the length of the `self` parameter. The method `concat` returns the string formed by concatenating `s` after `self`. The method `substr` returns the substring of its `self` parameter beginning at position `i` 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 `String` is `""` (not `void`). It is an error to inherit from or redefine `String`.

8.5 Bool

The `Bool` class provides `true` and `false`. The default initialization for variables of type `Bool` is `false` (not `void`). It is an error to inherit from or redefine `Bool`.

9 Main Class

Every program must have a class `Main`. Furthermore, the `Main` class must have a method `main` that takes no formal parameters. The `main` method must be defined in class `Main` (not inherited from another class). A program is executed by evaluating `(new Main).main()`.

The remaining sections of this manual provide a more formal definition of Cool. There are four sections covering lexical structure (Section 10), grammar (Section 11), type rules (Section 12), and operational semantics (Section 13).

10 Lexical Structure

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

10.1 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; object identifiers begin with a lower case letter. There are two other identifiers, **self** and **SELF_TYPE** that are treated specially by Cool but are not treated as keywords. The special syntactic symbols (e.g., parentheses, assignment operator, etc.) are given in Figure 1.

10.2 Strings

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

\b	backspace
\t	tab
\n	newline
\f	formfeed

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

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

A string may not contain EOF. A string may not contain the null (character \0). Any other character may be included in a string. Strings cannot cross file boundaries.

10.3 Comments

There are two forms of comments in Cool. Any characters between two dashes "`--`" 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 $(* \ldots *)$. The latter form of comment may be nested. Comments cannot cross file boundaries.

10.4 Keywords

The keywords of cool are: **class, else, false, fi, if, in, inherits, isvoid, let, loop, pool, then, while, case, esac, new, of, not, true.** Except for the constants **true** and **false**, keywords are case insensitive. To conform to the rules for other objects, the first letter of **true** and **false** must be lowercase; the trailing letters may be upper or lower case.

10.5 White Space

White space consists of any sequence of the characters: blank (ascii 32), `\n` (newline, ascii 10), `\f` (form feed, ascii 12), `\r` (carriage return, ascii 13), `\t` (tab, ascii 9), `\v` (vertical tab, ascii 11).

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{class};^{+} \\
\textit{class} & ::= & \textbf{class}\ \mathrm{TYPE}\ [\textbf{inherits}\ \mathrm{TYPE}]\ \{\ \textit{feature};^{*}\} \\
\textit{feature} & ::= & \mathrm{ID}(\textit{formal},^{*})\ :\ \mathrm{TYPE}\ \{\ \textit{expr}\ \} \\
 & | & \mathrm{ID}\ :\ \mathrm{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ] \\
\textit{formal} & ::= & \mathrm{ID}\ :\ \mathrm{TYPE} \\
\textit{expr} & ::= & \mathrm{ID}\ \texttt{<-}\ \textit{expr} \\
 & | & \textit{expr}[@\mathrm{TYPE}].\mathrm{ID}(\textit{expr},^{*}) \\
 & | & \mathrm{ID}(\textit{expr},^{*}) \\
 & | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{expr}\ \textbf{else}\ \textit{expr}\ \textbf{fi} \\
 & | & \textbf{while}\ \textit{expr}\ \textbf{loop}\ \textit{expr}\ \textbf{pool} \\
 & | & \{\ \textit{expr};^{+}\} \\
 & | & \textbf{let}\ [\![\mathrm{ID}\ :\ \mathrm{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ]]\!],^{+}\ \textbf{in}\ \textit{expr} \\
 & | & \textbf{case}\ \textit{expr}\ \textbf{of}\ [\![\mathrm{ID}\ :\ \mathrm{TYPE} => \textit{expr};]\!]^{+}\textbf{esac} \\
 & | & \textbf{new}\ \mathrm{TYPE} \\
 & | & \textbf{isvoid}\ \textit{expr} \\
 & | & \textit{expr} + \textit{expr} \\
 & | & \textit{expr} - \textit{expr} \\
 & | & \textit{expr} * \textit{expr} \\
 & | & \textit{expr}\,/\,\textit{expr} \\
 & | & \tilde{}\,\textit{expr} \\
 & | & \textit{expr} < \textit{expr} \\
 & | & \textit{expr} <= \textit{expr} \\
 & | & \textit{expr} = \textit{expr} \\
 & | & \textbf{not}\ \textit{expr} \\
 & | & (\textit{expr}) \\
 & | & \mathrm{ID} \\
 & | & \mathrm{integer} \\
 & | & \mathrm{string} \\
 & | & \textbf{true} \\
 & | & \textbf{false}
\end{array}
$$

Figure 1: Cool syntax.

11 Cool Syntax

Figure 1 provides a specification of Cool 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. The special notation $A;^*$ means zero or more A's terminated by semicolons. The special notation $A,^*$ means zero or more A's separated by commas. Separators differ from terminators in that the terminator always appears at the end of the list; a separator is never included at the end of a list. In Cool, semicolons are always terminators and commas are always separators. Items in square brackets [...] are optional. Double braces $[\![\,]\!]$ are not part of Cool; they are used in the grammar as a meta-symbol to show association of grammar symbols (e.g. $a[\![bc]\!]^+$ means a followed by one or more bc pairs).

11.1 Precedence

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

```
.
@
~
isvoid
* /
+ -
<=  <  =
not
<-
```

All binary operations are left-associative, with the exception of assignment, which is right-associative, and the three comparison operations, which do not associate.

12 Type Rules

This section formally defines the type rules of Cool. The type rules define the type of every Cool expression in a given context. The context is the *type environment*, which describes the type of every unbound identifier appearing in an expression. The type environment is described in Section 12.1. Section 12.2 gives the type rules.

12.1 Type Environments

To a first approximation, type checking in Cool can be thought of as a bottom-up algorithm: the type of an expression e is computed from the (previously computed) types of e's subexpressions. For example, an integer `1` has type `Int`; there are no subexpressions in this case. As another example, if $\mathtt{e_n}$ has type `X`, then the expression { $\mathtt{e_1}; \ldots ; \mathtt{e_n};$ } has type `X`.

A complication arises in the case of an expression `v`, where `v` is an object identifier. It is not possible to say what the type of `v` is in a strictly bottom-up algorithm; we need to know the type declared for `v` in the larger expression. Such a declaration must exist for every object identifier in valid Cool programs.

To capture information about the types of identifiers, we use a *type environment*. The environment consists of three parts: a method environment M, an object environment O, and the name of the current class in which the expression appears. The method environment and object environment are both functions (also called *mappings*). The object environment is a function of the form

$$O(v) = T$$

which assigns the type T to object identifier v. The method environment is more complex; it is a function of the form

$$M(C, f) = (T_1, \ldots, T_{n-1}, T_n)$$

where C is a class name (a type), f is a method name, and $t_1, \ldots, t_n$ are types. The tuple of types is the *signature* of the method. The interpretation of signatures is that in class C the method f has formal parameters of types $(t_1, \ldots, t_{n-1})$—in that order—and a return type t_n.

Two mappings are required instead of one because object names and method names do not clash—i.e., there may be a method and an object identifier of the same name.

The third component of the type environment is the name of the current class, which is needed for type rules involving `SELF_TYPE`.

Every expression e is type checked in a type environment; the subexpressions of e may be type checked in the same environment or, if e introduces a new object identifier, in a modified environment. For example, consider the expression

```
let c : Int <- 33 in
  ...
```

The `let` expression introduces a new variable `c` with type `Int`. Let O be the object component of the type environment for the `let`. Then the body of the `let` is type checked in the object type environment

$$O[Int/c]$$

where the notation $O[T/c]$ is defined as follows:

$$\begin{array}{rcl} O[T/c](c) & = & T \\ O[T/c](d) & = & O(d) \;\; \text{if } d \neq c \end{array}$$

12.2 Type Checking Rules

The general form a type checking rule is:

$$\frac{\vdots}{O, M, C \vdash e : T}$$

The rule should be read: In the type environment for objects O, methods M, and containing class C, the expression e has type T. The dots above the horizontal bar stand for other statements about the types of sub-expressions of e. These other statements are hypotheses of the rule; if the hypotheses are satisfied, then the statement below the bar is true. In the conclusion, the "turnstyle" ("$\vdash$") separates context (O, M, C) from statement $(e : T)$.

The rule for object identifiers is simply that if the environment assigns an identifier Id type T, then Id has type T.

$$\frac{O(Id) = T}{O, M, C \vdash Id : T} \qquad \text{[Var]}$$

The rule for assignment to a variable is more complex:

$$\frac{\begin{array}{l} O(Id) = T \\ O, M, C \vdash e_1 : T' \\ T' \leq T \end{array}}{O, M, C \vdash Id \leftarrow e_1 : T'} \quad \text{[ASSIGN]}$$

Note that this type rule—as well as others—use the conformance relation $\leq$ (see Section 3.2). The rule says that the assigned expression e_1 must have a type T' that conforms to the type T of the identifier Id in the type environment. The type of the whole expression is T'.

The type rules for constants are all easy:

$$\frac{}{O, M, C \vdash true : Bool} \quad \text{[True]}$$

$$\frac{}{O, M, C \vdash false : Bool} \quad \text{[False]}$$

$$\frac{i \text{ is an integer constant}}{O, M, C \vdash i : Int} \quad \text{[Int]}$$

$$\frac{s \text{ is a string constant}}{O, M, C \vdash s : String} \quad \text{[String]}$$

There are two cases for `new`, one for `new SELF_TYPE` and one for any other form:

$$\frac{T' = \left\{ \begin{array}{rl} \texttt{SELF\_TYPE}_C & \text{if } T = \texttt{SELF\_TYPE} \\ T & \text{otherwise} \end{array} \right.}{O, M, C \vdash new\ T : T'} \quad \text{[New]}$$

Dispatch expressions are the most complex to type check.

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0' = \left\{ \begin{array}{ll} C & \text{if } T_0 = \texttt{SELF\_TYPE}_C \\ T_0 & \text{otherwise} \end{array} \right. \\ M(T_0', f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ T_{n+1} = \left\{ \begin{array}{rl} T_0 & \text{if } T_{n+1}' = \texttt{SELF\_TYPE} \\ T_{n+1}' & \text{otherwise} \end{array} \right. \end{array}}{O, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}} \quad \text{[Dispatch]}$$

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0 \leq T \\ M(T, f) = (T'_1, \ldots, T'_n, T'_{n+1}) \\ T_i \leq T'_i \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T'_{n+1} = \texttt{SELF\_TYPE} \\ T'_{n+1} & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}} \quad \text{[StaticDispatch]}$$

To type check a dispatch, each of the subexpressions must first be type checked. The type T_0 of e_0 determines which declaration of the method f is used. The argument types of the dispatch must conform to the declared argument types. Note that the type of the result of the dispatch is either the declared return type or T_0 in the case that the declared return type is `SELF_TYPE`. The only difference in type checking a static dispatch is that the class T of the method f is given in the dispatch, and the type T_0 must conform to T.

The type checking rules for `if` and {-} expressions are straightforward. See Section 7.5 for the definition of the $\sqcup$ operation.

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : Bool \\ O, M, C \vdash e_2 : T_2 \\ O, M, C \vdash e_3 : T_3 \end{array}}{O, M, C \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : T_2 \sqcup T_3} \quad \text{[If]}$$

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ \vdots \\ O, M, C \vdash e_n : T_n \end{array}}{O, M, C \vdash \{\ e_1; e_2; \ldots e_n;\ \} : T_n} \quad \text{[Sequence]}$$

The `let` rule has some interesting aspects.

$$\frac{\begin{array}{l} T'_0 = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leq T'_0 \\ O[T'_0/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{let } x : T_0 \leftarrow e_1 \text{ in } e_2 : T_2} \quad \text{[Let-Init]}$$

First, the initialization e_1 is type checked in an environment without a new definition for x. Thus, the variable x cannot be used in e_1 unless it already has a definition in an outer scope. Second, the body e_2 is type checked in the environment O extended with the typing $x : T'_0$. Third, note that the type of x may be `SELF_TYPE`.

$$
\begin{array}{c}
T_0' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\
O[T_0'/x], M, C \vdash e_1 : T_1 \\
\hline
O, M, C \vdash \text{let } x : T_0 \text{ in } e_1 : T_1
\end{array} \qquad \text{[Let-No-Init]}
$$

The rule for `let` with no initialization simply omits the conformance requirement. We give type rules only for a `let` with a single variable. Typing a multiple `let`

$$
\mathrm{let}\ \mathrm{x_1} : \mathrm{T_1}\ [\leftarrow \mathrm{e_1}],\ \mathrm{x_2} : \mathrm{T_2}\ [\leftarrow \mathrm{e2}], \ldots,\ \mathrm{x_n} : \mathrm{T_n}\ [\leftarrow \mathrm{e_n}]\ \mathrm{in\ e}
$$

is defined to be the same as typing

$$
\mathrm{let}\ \mathrm{x_1} : \mathrm{T_1}\ [\leftarrow \mathrm{e_1}]\ \mathrm{in}\ (\mathrm{let}\ \mathrm{x_2} : \mathrm{T_2}\ [\leftarrow \mathrm{e_2}], \ldots,\ \mathrm{x_n} : \mathrm{T_n}\ [\leftarrow \mathrm{e_n}]\ \mathrm{in\ e}\ )
$$

$$
\begin{array}{c}
O, M, C \vdash e_0 : T_0 \\
O[T_1/x_1], M, C \vdash e_1 : T_1' \\
\vdots \\
O[T_n/x_n], M, C \vdash e_n : T_n' \\
\hline
O, M, C \vdash \text{case } e_0 \text{ of } x_1 : T_1 \Rightarrow e_1; \ldots x_n : T_n \Rightarrow e_n;\ \text{esac} : \bigsqcup_{1 \le i \le n} T_i'
\end{array} \qquad \text{[Case]}
$$

Each branch of a `case` is type checked in an environment where variable x_i has type T_i. The type of the entire `case` is the join of the types of its branches. The variables declared on each branch of a `case` must all have distinct types.

$$
\begin{array}{c}
O, M, C \vdash e_1 : Bool \\
O, M, C \vdash e_2 : T_2 \\
\hline
O, M, C \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : Object
\end{array} \qquad \text{[Loop]}
$$

The predicate of a loop must have type $Bool$; the type of the entire loop is always $Object$. An `isvoid` test has type $Bool$:

$$
\begin{array}{c}
O, M, C \vdash e_1 : T_1 \\
\hline
O, M, C \vdash \text{isvoid } e_1 : Bool
\end{array} \qquad \text{[Isvoid]}
$$

With the exception of the rule for equality, the type checking rules for the primitive logical, comparison, and arithmetic operations are easy.

$$
\begin{array}{c}
O, M, C \vdash e_1 : Bool \\
\hline
O, M, C \vdash \neg e_1 : Bool
\end{array} \qquad \text{[Not]}
$$

$$
\begin{array}{c}
O, M, C \vdash e_1 : Int \\
O, M, C \vdash e_2 : Int \\
op \in \{<, \le\} \\
\hline
O, M, C \vdash e_1\ op\ e_2 : Bool
\end{array} \qquad \text{[Compare]}
$$

$$
\begin{array}{c}
O, M, C \vdash e_1 : Int \\
\hline
O, M, C \vdash \tilde{}e_1 : Int
\end{array} \qquad \text{[Neg]}
$$

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Int \\ O, M, C \vdash e_2 : Int \\ op \in \{*, +, -, /\} \end{array}}{O, M, C \vdash e_1 \; op \; e_2 : Int} \quad \text{[Arith]}$$

The wrinkle in the rule for equality is that any types may be freely compared except `Int`, `String` and `Bool`, which may only be compared with objects of the same type.

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{Int, String, Bool\} \vee T_2 \in \{Int, String, Bool\} \Rightarrow T_1 = T_2 \end{array}}{O, M, C \vdash e_1 = e_2 : Bool} \quad \text{[Equal]}$$

The final cases are type checking rules for attributes and methods. For a class C, let the object environment O_C give the types of all attributes of C (including any inherited attributes). More formally, if x is an attribute (inherited or not) of C, and the declaration of x is $x : T$, then

$$O_C(x) = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T = \texttt{SELF\_TYPE} \\ T & \text{otherwise} \end{cases}$$

The method environment M is global to the entire program and defines for every class C the signatures of all of the methods of C (including any inherited methods).

The two rules for type checking attribute defininitions are similar the rules for `let`. The essential difference is that attributes are visible within their initialization expressions. Note that `self` is bound in the initialization.

$$\frac{\begin{array}{l} O_C(x) = T_0 \\ O_C[\texttt{SELF\_TYPE}_C/self], M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C, M, C \vdash x : T_0 \leftarrow e_1;} \quad \text{[Attr-Init]}$$

$$\frac{O_C(x) = T}{O_C, M, C \vdash x : T;} \quad \text{[Attr-No-Init]}$$

The rule for typing methods checks the body of the method in an environment where O_C is extended with bindings for the formal parameters and `self`. The type of the method body must conform to the declared return type.

$$\frac{\begin{array}{l} M(C, f) = (T_1, \ldots, T_n, T_0) \\ O_C[\texttt{SELF\_TYPE}_C/self][T_1/x_1] \ldots [T_n/x_n], M, C \vdash e : T_0' \\ T_0' \leq \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \end{array}}{O_C, M, C \vdash f(x_1 : T_1, \ldots, x_n : T_n) : T_0 \; \{ \; e \; \};} \quad \text{[Method]}$$

13 Operational Semantics

This section contains a mostly formal presentation of the operational semantics for the Cool language. The operational semantics define for every Cool expression what value it should produce in a given context. The context has three components: an environment, a store, and a self object. These components are described in the next section. Section 13.2 defines the syntax used to refer to Cool objects, and Section 13.3 defines the syntax used to refer to class definitions.

Keep in mind that a formal semantics is a specification only—it does not describe an implementation. The purpose of presenting the formal semantics is to make clear all the details of the behavior of Cool expressions. How this behavior is implemented is another matter.

13.1 Environment and the Store

Before we can present a semantics for Cool we need a number of concepts and a considerable amount of notation. An *environment* is a mapping of variable identifiers to *locations*. Intuitively, an environment tells us for a given identifier the address of the memory location where that identifier's value is stored. For a given expression, the environment must assign a location to all identifiers to which the expression may refer. For the expression, e.g., $a + b$, we need an environment that maps a to some location and b to some location. We'll use the following syntax to describe environments, which is very similar to the syntax of type assumptions used in Section 12.

$$E = [a : l_1, b : l_2]$$

This environment maps a to location l_1, and b to location l_2.

The second component of the context for the evaluation of an expression is the *store* (memory). The store maps locations to values, where values in Cool are just objects. Intuitively, a store tells us what value is stored in a given memory location. For the moment, assume all values are integers. A store is similar to an environment:

$$S = [l_1 \rightarrow 55, l_2 \rightarrow 77]$$

This store maps location l_1 to value 55 and location l_2 to value 77.

Given an environment and a store, the value of an identifier can be found by first looking up the location that the identifier maps to in the environment and then looking up the location in the store.

$$\begin{array}{rcl} E(a) & = & l_1 \\ S(l_1) & = & 55 \end{array}$$

Together, the environment and the store define the execution state at a particular step of the evaluation of a Cool expression. The double indirection from identifiers to locations to values allows us to model variables. Consider what happens if the value 99 is assigned variable a in the environment and store defined above. Assigning to a variable means changing the value to which it refers but not its location. To perform the assignment, we look up the location for a in the environment E and then change the mapping for the obtained location to the new value, giving a new store S'.

$$\begin{array}{rcl} E(a) & = & l_1 \\ S' & = & S[99/l_1] \end{array}$$

The syntax $S[v/l]$ denotes a new store that is identical to the store S, except that S' maps location l to value v. For all locations l' where $l' \neq l$, we still have $S'(l') = S(l')$.

The store models the contents of memory of the computer during program execution. Assigning to a variable modifies the store.

There are also situations in which the environment is modified. Consider the following Cool fragment:

```
let c : Int <- 33 in
  c
```

When evaluating this expression, we must introduce the new identifier c into the environment before evaluating the body of the `let`. If the current environment and state are E and S, then we create a new environment E' and a new store S' defined by:

$$\begin{aligned} l_c &= newloc(S) \\ E' &= E[l_c/c] \\ S' &= S[33/l_c] \end{aligned}$$

The first step is to allocate a location for the variable c. The location should be fresh, meaning that the current store does not have a mapping for it. The function $newloc()$ applied to a store gives us an unused location in that store. We then create a new environment E', which maps c to l_c but also contains all of the mappings of E for identifiers other than c. Note that if c already has a mapping in E, the new environment E' hides this old mapping. We must also update the store to map the new location to a value. In this case l_c maps to the value 33, which is the initial value for c as defined by the let-expression.

The example in this subsection oversimplifies Cool environments and stores a bit, because simple integers are not Cool values. Even integers are full-fledged objects in Cool.

13.2 Syntax for Cool Objects

Every Cool value is an object. Objects contain a list of named attributes, a bit like records in C. In addition, each object belongs to a class. We use the following syntax for values in Cool:

$$v = X(a_1 = l_1, a_2 = l_2, \ldots, a_n = l_n)$$

Read the syntax as follows: The value v is a member of class X containing the attributes $a_1, \ldots, a_n$ whose locations are $l_1, \ldots, l_n$. Note that the attributes have an associated location. Intuitively this means that there is some space in memory reserved for each attribute.

For base objects of Cool (i.e., `Int`s, `String`s, and `Bool`s) we use a special case of the above syntax. Base objects have a class name, but their attributes are not like attributes of normal classes, because they cannot be modified. Therefore, we describe base objects using the following syntax:

$$\begin{aligned} &Int(5) \\ &Bool(true) \\ &String(4, \texttt{"}Cool\texttt{"}) \end{aligned}$$

For `Int`s and `Bool`s, the meaning is obvious. `String`s contain two parts, the length and the actual sequence of ASCII characters.

13.3 Class definitions

In the rules presented in the next section, we need a way to refer to the definitions of attributes and methods for classes. Suppose we have the following Cool class definition:

```
class B {
   s : String <- "Hello";
   g (y:String) : Int {
      y.concat(s)
   };
   f (x:Int) : Int {
```

```
      x+1
   };
};

class A inherits B {
   a : Int;
   b : B <- new B;
   f(x:Int) : Int {
      x+a
   };
};
```

Two mappings, called *class* and *implementation*, are associated with class definitions. The *class* mapping is used to get the attributes, as well as their types and initializations, of a particular class:

$$class(A) = (s : String \leftarrow \texttt{"}Hello\texttt{"},\ a : Int \leftarrow 0,\ b : B \leftarrow new\ B)$$

Note that the information for class A contains everything that it inherited from class B, as well as its own definitions. If B had inherited other attributes, those attributes would also appear in the information for A. The attributes are listed in the order they are inherited and then in source order: all the attributes from the greatest ancestor are listed first in the order in which they textually appear, then the attributes of the next greatest ancestor, and so on, on down to the attributes defined in the particular class. We rely on this order in describing how new objects are initialized.

The general form of a class mapping is:

$$class(X) = (a_1 : T_1 \leftarrow e_1,\ \ldots,\ a_n : T_n \leftarrow e_n)$$

Note that every attribute has an initializing expression, even if the Cool program does not specify one for each attribute. The default initialization for a variable or attribute is the *default* of its type. The default of `Int` is 0, the default of `String` is `""`, the default of `Bool` is `false`, and the default of any other type is `void`.[4] The default of type T is written D_T.

The implementation mapping gives information about the methods of a class. For the above example, *implementation* of A is defined as follows:

$$\begin{array}{rcl} implementation(A, f) & = & (x, x + a) \\ implementation(A, g) & = & (y, y.concat(s)) \end{array}$$

In general, for a class X and a method m,

$$implementation(X, m) = (x_1, x_2, \ldots, x_n, e_{body})$$

specifies that method m when invoked from class X, has formal parameters $x_1, \ldots, x_n$, and the body of the method is expression e_{body}.

[4] A tiny point: We are allowing `void` to be used as an expression here. There is no expression for `void` available to Cool programmers.

13.4 Operational Rules

Equipped with environments, stores, objects, and class definitions, we can now attack the operational semantics for Cool. The operational semantics is described by rules similar to the rules used in type checking. The general form of the rules is:

$$\frac{\vdots}{so, S, E \vdash e_1 : v, S'}$$

The rule should be read as: In the context where *self* is the object so, the store is S, and the environment is E, the expression e_1 evaluates to object v and the new store is S'. The dots above the horizontal bar stand for other statements about the evaluation of sub-expressions of e_1.

Besides an environment and a store, the evaluation context contains a self object so. The self object is just the object to which the identifier `self` refers if `self` appears in the expression. We do not place `self` in the environment and store because `self` is not a variable—it cannot be assigned to. Note that the rules specify a new store after the evaluation of an expression. The new store contains all changes to memory resulting as side effects of evaluating expression e_1.

The rest of this section presents and briefly discusses each of the operational rules. A few cases are not covered; these are discussed at the end of the section.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ E(Id) = l_1 \\ S_3 = S_2[v_1/l_1] \end{array}}{so, S_1, E \vdash Id \leftarrow e_1 : v_1, S_3} \quad \text{[Assign]}$$

An assignment first evaluates the expression on the right-hand side, yielding a value v_1. This value is stored in memory at the address for the identifier.

The rules for identifier references, `self`, and constants are straightforward:

$$\frac{\begin{array}{c} E(Id) = l \\ S(l) = v \end{array}}{so, S, E \vdash Id : v, S} \quad \text{[Var]}$$

$$\frac{}{so, S, E \vdash \text{self} : so, S} \quad \text{[Self]}$$

$$\frac{}{so, S, E \vdash \text{true} : Bool(true), S} \quad \text{[True]}$$

$$\frac{}{so, S, E \vdash \text{false} : Bool(false), S} \quad \text{[False]}$$

$$\frac{i \text{ is an integer constant}}{so, S, E \vdash i : Int(i), S} \quad \text{[Int]}$$

$$\frac{\begin{array}{l} s \text{ is a string constant} \\ l = length(s) \end{array}}{so, S, E \vdash s : String(l, s), S} \quad \text{[String]}$$

A `new` expression is more complicated than one might expect:

$$\frac{\begin{array}{l} T_0 = \begin{cases} X & \text{if } T = \texttt{SELF\_TYPE} \text{ and } so = X(\ldots) \\ T & \text{otherwise} \end{cases} \\ class(T_0) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \\ l_i = newloc(S_1), \text{ for } i = 1 \ldots n \text{ and each } l_i \text{ is diistinct} \\ v_1 = T_0(a_1 = l_1, \ldots, a_n = l_n) \\ S_2 = S_1[D_{T_1}/l_1, \ldots, D_{T_n}/l_n] \\ v_1, S_2, [a_1 : l_1, \ldots, a_n : l_n] \vdash \{a_1 \leftarrow e_1; \ldots; a_n \leftarrow e_n;\} : v_2, S_3 \end{array}}{so, S_1, E \vdash \text{new } T : v_1, S_3} \quad \text{[New]}$$

The tricky thing in a `new` expression is to initialize the attributes in the right order. Note also that, during initialization, attributes are bound to the default of the appropriate class.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n : v_n, S_{n+1} \\ so, S_{n+1}, E \vdash e_0 : v_0, S_{n+2} \\ v_0 = X(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \\ implementation(X, f) = (x_1, \ldots, x_n, e_{n+1}) \\ l_{x_i} = newloc(S_{n+2}), \text{ for } i = 1 \ldots n \text{ and each } l_{x_i} \text{ is distinct} \\ S_{n+3} = S_{n+2}[v_1/l_{x_1}, \ldots, v_n/l_{x_n}] \\ v_0, S_{n+3}, [a_1 : l_{a_1}, \ldots, a_m : l_{a_m}, x_1 : l_{x_1}, \ldots, x_n : l_{x_n}] \vdash e_{n+1} : v_{n+1}, S_{n+4} \end{array}}{so, S_1, E \vdash e_0.f(e_1, \ldots, e_n) : v_{n+1}, S_{n+4}} \quad \text{[Dispatch]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n : v_n, S_{n+1} \\ so, S_{n+1}, E \vdash e_0 : v_0, S_{n+2} \\ v_0 = X(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \\ implementation(T, f) = (x_1, \ldots, x_n, e_{n+1}) \\ l_{x_i} = newloc(S_{n+2}), \text{ for } i = 1 \ldots n \text{ and each } l_{x_i} \text{ is distinct} \\ S_{n+3} = S_{n+2}[v_1/l_{x_1}, \ldots, v_n/l_{x_n}] \\ v_0, S_{n+3}, [a_1 : l_{a_1}, \ldots, a_m : l_{a_m}, x_1 : l_{x_1}, \ldots, x_n : l_{x_n}] \vdash e_{n+1} : v_{n+1}, S_{n+4} \end{array}}{so, S_1, E \vdash e_0@T.f(e_1, \ldots, e_n) : v_{n+1}, S_{n+4}} \quad \text{[StaticDispatch]}$$

The two dispatch rules do what one would expect. The arguments are evaluated and saved. Next, the expression on the left-hand side of the "." is evaluated. In a normal dispatch, the class of this expression is used to determine the method to invoke; otherwise the class is specified in the dispatch itself.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(true), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \end{array}}{so, S_1, E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : v_2, S_3} \quad \text{[If-True]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(false), S_2 \\ so, S_2, E \vdash e_3 : v_3, S_3 \end{array}}{so, S_1, E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : v_3, S_3} \quad \text{[If-False]}$$

There are no surprises in the if-then-else rules. Note that value of the predicate is a `Bool` object, not a boolean.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ \vdots \\ so, S_n, E \vdash e_n : v_n, S_{n+1} \end{array}}{so, S_1, E \vdash \{\ e_1; e_2; \ldots; e_n;\ \} : v_n, S_{n+1}} \quad \text{[Sequence]}$$

Blocks are evaluated from the first expression to the last expression, in order. The result is the result of the last expression.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : v_1, S_2 \\ l_1 = newloc(S_2) \\ S_3 = S_2[v_1/l_1] \\ E' = E[l_1/Id] \\ so, S_3, E' \vdash e_2 : v_2, S_4 \end{array}}{so, S_1, E \vdash \text{let } Id : T_1 \leftarrow e_1 \text{ in } e_2 : v_2, S_4} \quad \text{[Let]}$$

A `let` evaluates any initialization code, assigns the result to the variable at a fresh location, and evaluates the body of the `let`. (If there is no initialization, the variable is initialized to the default value of T_1.) We give the operational semantics only for the case of `let` with a single variable. The semantics of a multiple `let`

$$\text{let } \mathrm{x_1} : \mathrm{T_1} \leftarrow \mathrm{e_1},\ \mathrm{x_2} : \mathrm{T_2} \leftarrow \mathrm{e2}, \ldots,\ \mathrm{x_n} : \mathrm{T_n} \leftarrow \mathrm{e_n} \text{ in e}$$

is defined to be the same as

$$\text{let } \mathrm{x_1} : \mathrm{T_1} \leftarrow \mathrm{e_1} \text{ in (let } \mathrm{x_2} : \mathrm{T_2} \leftarrow \mathrm{e_2}, \ldots,\ \mathrm{x_n} : \mathrm{T_n} \leftarrow \mathrm{e_n} \text{ in e)}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_0 : v_0, S_2 \\ v_0 = X(\ldots) \\ T_i = \text{closest ancestor of } X \text{ in } \{T_1, \ldots, T_n\} \\ l_0 = newloc(S_2) \\ S_3 = S_2[v_0/l_0] \\ E' = E[l_0/Id_i] \\ so, S_3, E' \vdash e_i : v_1, S_4 \end{array}}{so, S_1, E \vdash \text{case } e_0 \text{ of } Id_1 : T_1 \Rightarrow e_1; \ldots; Id_n : T_n \Rightarrow e_n;\ \text{esac} : v_1, S_4} \quad \text{[Case]}$$

Note that the `case` rule requires that the class hierarchy be available in some form at runtime, so that the correct branch of the `case` can be selected. This rule is otherwise straightforward.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(true), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ so, S_3, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : void, S_4 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : void, S_4} \quad \text{[Loop-True]}$$

$$\frac{so, S_1, E \vdash e_1 : Bool(false), S_2}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : void, S_2} \quad \text{[Loop-False]}$$

There are two rules for `while`: one for the case where the predicate is `true` and one for the case where the predicate is `false`. Both cases are straightforward. The two rules for `isvoid` are also straightforward:

$$\frac{so, S_1, E \vdash e_1 : void, S_2}{so, S_1, E \vdash \text{isvoid } e_1 : Bool(true), S_2} \quad \text{[IsVoid-True]}$$

$$\frac{so, S_1, E \vdash e_1 : X(\ldots), S_2}{so, S_1, E \vdash \text{isvoid } e_1 : Bool(false), S_2} \quad \text{[IsVoid-False]}$$

The remainder of the rules are for the primitive arithmetic, logical, and comparison operations except equality. These are all easy rules.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(b), S_2 \\ v_1 = Bool(\neg b) \end{array}}{so, S_1, E \vdash \text{not } e_1 : v_1, S_2} \quad \text{[Not]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Int(i_1), S_2 \\ so, S_2, E \vdash e_2 : Int(i_2), S_3 \\ op \in \{\leq, <\} \\ v_1 = \begin{cases} Bool(true), & \text{if } i_1 \ op \ i_2 \\ Bool(false), & \text{otherwise} \end{cases} \end{array}}{so, S_1, E \vdash e_1 \ op \ e_2 : v_1, S_3} \quad \text{[Comp]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Int(i_1), S_2 \\ v_1 = Int(-i_1) \end{array}}{so, S_1, E \vdash \tilde{\ } e_1 : v_1, S_2} \quad \text{[Neg]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Int(i_1), S_2 \\ so, S_2, E \vdash e_2 : Int(i_2), S_3 \\ op \in \{*, +, -, /\} \\ v_1 = Int(i_1 \ op \ i_2) \end{array}}{so, S_1, E \vdash e_1 \ op \ e_2 : v_1, S_3} \quad \text{[Arith]}$$

Cool `Int`s are 32-bit two's complement signed integers; the arithmetic operations are defined accordingly.

The notation and rules given above are not powerful enough to describe how objects are tested for equality, or how runtime exceptions are handled. For these cases we resort to an English description.

In $e_1 = e_2$, first e_1 is evaluated and then e_2 is evaluated. The two objects are compared for equality by first comparing their pointers (addresses). If they are the same, the objects are equal. The value `void` is not equal to any object except itself. If the two objects are of type `String`, `Bool`, or `Int`, their respective contents are compared.

In addition, the operational rules do not specify what happens in the event of a runtime error. Execution aborts when a runtime error occurs. The following list specifies all possible runtime errors.

1. A dispatch (static or dynamic) on `void`.

2. A case on `void`.

3. Execution of a case statement without a matching branch.

4. Division by zero.

5. Substring out of range.

6. Heap overflow.

Finally, the rules given above do not explain the execution behaviour for dispatches to primitive methods defined in the `Object`, `IO`, or `String` classes. Descriptions of these primitive methods are given in Sections 8.3-8.5.

14 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 current version.