CoolAid: The Cool 2020 Reference Manual*

1 Introduction

This manual describes the programming language Cool: the Classroom Object-Oriented Language. Cool is a small language that can be implemented 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 2020 is a new language differing from many versions of Cool. Indeed Cool 2020 is a subset of Scala; in a ridiculous act of hubris, we will call Scala "Extended Cool." Part of this class involves learning to use a new language quickly.

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 ten to eleven pages (through Section 9) suffices. The formal description begins with Section 10.

In this manual, $CLASSHOME is assumed to be `/afs/cs.uwm.edu/users/classes/cs654/`. We also assume you have `$CLASSHOME/cmd` in your UNIX "path."

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 `$CLASSHOME/examples`. Cool source files have extension `.cool` and Cool assembly files have extension `.s`. The Cool compiler is `$CLASSHOME/cmd/coolc`. To compile a program:

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

The compiler compiles the files `file1.cool` through `filen.cool` 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 we don't have a MIPS-based machine to run them on, Cool programs are run on a MIPS simulator called `spim`. We need to include the runtime

---

system. For convenience, we provide a shell script `coolspim` that calls `spim` with the right arguments. To run a cool program, type

```
% coolspim
(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:

```
coolspim file.s
```

This form loads the file, runs the program, and exits `spim` when the program terminates. The `spim` manual is available on the course Web page.

The following is a complete transcript of the compilation and execution of `sort_list.cool` from the examples directory `$CLASSHOME/examples/`. This program is very silly, but it does serve to illustrate many of the features of Cool.

```
% coolc sort_list.cool
% coolspim sort_list.s
SPIM Version 8.0 of January 8, 2010 (with STATS by jtb)
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /afs/cs.uwm.edu/users/classes/cs654/lib/cool-runtime.s
How many numbers to sort? 5
0
1
2
3
4
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(var-formals) [ extends type(actuals) ] {
     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. The formal parameters are attributes (see next section) initialized when instances are created.

2

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. A special method in each class is the *constructor*, which has the same name as the class (and no declared return type). The constructor is used to initialize newly created objects of the class.

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 inspired by `sort_list.cool` illustrates simple cases of both attributes and methods:

```
class Cons(var car : Int, var cdr : List) extends List() {
  override def isNil() : Boolean = false;

  override def head() : Int = car;

  override def tail() : List = cdr;
}
```

In this example, the class `Cons` has two attributes `car` and `cdr` (both of which are initialized when instances are created) and three methods `isNil`, `head` and `tail`, each of which is declared as overriding a method in the parent class. The types of attributes, as well as the types of formal parameters and return types of methods are explicitly declared by the programmer.

We can create an object of class `Cons` and set the `car` and `cdr` fields in the following example:

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

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. 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.

Given an object `c` of class `Cons`, we can access the `car` field using the syntax:

```
c.head()
```

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

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

---

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

3.2 Inheritance

If a class definition has the form

```
class A(...) extends B(...)  { ... }
```

then class A inherits the features of B. In this case B is the *parent* class of A and A is a *child* class of B.

The semantics of A inheriting from B is that A has all of the features defined in B 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. For type safety, it is necessary to place some restrictions on how methods may be redefined (see Section 6). It is illegal to redefine attribute names.

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 A inherits from B, then B must not inherit from A, directly or indirectly. Furthermore, if A inherits from B, then B must have a class definition somewhere in the program.

There is a distinguished class `Any`. If some other class definition does not specify a parent class, then the class inherits from `Any` by default. Because Cool has single inheritance, the inheritance graph forms a tree with `Any` as the root. In addition to `Any`, Cool has several other *basic classes*: `Unit`, `Int`, `String`, `Boolean`, `ArrayAny`, `IO` and `Symbol`. Of these additional basic classes, only `IO` may be used as a superclass. The basic classes are defined in `$CLASSHOME/lib/basic.cool` and are documented there.

4 Types

In Cool, every class name is also a type. There are two built-in types that are not classes: `Null` and `Nothing`. A *value 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 it is an attribute, a `var`, a `case`, or as the formal parameter of a method. The type must be one of the two built-in types or a declared class.

The basic type rule in Cool is that if a method or variable expects a value of type A, then any value of type B may be used instead, provided that A is an ancestor of B in the class hierarchy. In other words, if B inherits from A, either directly or indirectly, then a B can be used wherever an A would suffice. The `Nothing` and `Null` types are handled specially.

When an object of class B may be used in place of an object of class A, we say that B *conforms* to A or that $B \le A$. As discussed above, conformance is defined in terms of the inheritance graph.

Definition 4.1 (Conformance) Let $A, B,$ and C be types.

- $A \le A$ for all types A

- if A inherits from B, then $A \le B$

- if $A \le B$ and $B \le C$ then $A \le C$

Because `Any` is the root of the class hierarchy, it follows that $A \le$ `Any` for all types A. The `Nothing` type conforms to all types; hence `Nothing` $\le A$ for all types A. The `Null` type conforms to all types *except* the three value classes: `Boolean`, `Int` and `Unit`.

---

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

4.1 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 D_e be a dynamic type of e and let S_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 $D_e \leq S_e$.

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

var *id* : *type* = *initial*;

(the type must not be the special type `Nothing`). An attribute is given a default initial value (see below) when the object is first created; the specified initialization is performed in the constructor (see Section 6.1).

Attributes may be accessed inside the class that defines them or in a class that inherits (perhaps indirectly) from this class. Inherited attributes cannot be redefined.

5.1 Null

Attributes in Cool are pre-initialized to contain values of the appropriate type. The special value `null` is used for attributes of all types except `Boolean`, `Unit` and `Int`. Attributes of the latter types are initialized by default to `false`, `()`, and `0` respectively.

The value `null` (which is the only value of type `Null`) cannot be safely tested using `==`, because this operator is converted into a method dispatch (q.v.) which can then raise a null dispatch error. It *can* be safely tested using `match` (q.v.). The predefined method `IO.is_null` uses this technique to test whether its argument is null. Null 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 `null` generates a runtime error.

There are *no* values of type `Nothing`.

6   Methods

A method definition has the form

[override] def *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 value of the method invocation. A formal parameter is allowed to shadow an attribute of the same name, even though this is often an error.

To ensure type safety, there are restrictions on the redefinition of inherited methods. If a class A inherits a method f from an ancestor class B, then A may override the inherited definition of f provided the number of arguments, and the types of the formal parameters must be exactly the same in both definitions, and the return type of the overriding method must conform to the return type of the overridden method.

Additionally, A must signal its intention to override with the `override` keyword. It is an error to unintentionally override a method (without `override`), or to claim to override a method when there is no method to override.

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

```
class A() {
    def f(): Int = 1;
}

class B() extends A() {
    override def f(): String = "1";
}
```

Let `a` be an object with dynamic type `A`. Then

```
a.f() + 1
```

is a well-formed expression with value 2. However, we cannot substitute a value of type `B` for `a`, 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.

6.1   Constructors

Each class has an implicit constructor that takes the class parameters as arguments, calls the superclass constructor and then performs attribute initialization. Additionally, a class may be include *initializers* of the form { ... }; in its body which are added to the (implicit) constructor. Initializers and attribute initializations are executed in the order in which they appear.

A constructor cannot be called as an ordinary method; it can only be called in the context of an allocation, such as

```
new Matrix(5,7)
```

7   Expressions

Expressions are the largest syntactic category in Cool.

7.1   Literals

The simplest expressions are literals. The unit literal is written `()`. The boolean literals are `true` and `false`. An integer literal is either the single digit `0` or a string of digits *not* starting with zero. So `100` is a legal integer, but `011` is not. String literals are sequences of characters enclosed in double quotes, such as `"This is a string."` There are other restrictions on strings; see Section 10.

   The literals belong to the basic classes `Boolean`, `Int`, `Unit` and `String`. The value of a literal is an object of the appropriate basic class. Additionally, the value `null` has the type of any class *except* the three value classes (`Unit`, `Boolean` and `Int`).

7.2   Identifiers

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

   Local variables and formal parameters have lexical scope. Attributes are visible throughout a class in which they are declared or inherited. The binding of an identifier reference is the innermost scope that contains a declaration for that identifier, or to the attribute of the same name if there is no other declaration. The exception to this rule is the identifier `this`, which is implicitly bound in every class. Cool 2020 forbids some forms of "shadowing" in which an inner variable has the same name as an outer variable. This avoids common programming errors.

7.3   Assignment

An assignment has the form

id `=` *expr*

Only attributes and local variables may be assigned. The static type of the expression must conform to the declared type of the identifier. The value of an assignment is `()`: the unit value.

7.4   Dispatch

There are three forms of dispatch 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, we evaluate the subexpressions in left-to-right order, from e_0 to e_n. Next, the class C of e_0 is noted (if e_0 is `null` a runtime error is generated). Finally, the method f in class C is invoked, with the value of e_0 bound to `this` in the body of f and the actual arguments bound to the formals as usual. The value of the expression is the value returned by the method invocation.

   Type checking a dispatch involves several steps. Assume e_0 has static type A. (Recall that this type is not necessarily the same as the type C above. A is the type inferred by the type checker; C is the class

of the object computed at 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.

The other forms of dispatch are:

$id\,(expr\,,\ldots,expr\,)$
`super.id(`$expr\,,\ldots,expr\,$`)`

The first form is shorthand for `this.`$id\,(expr\,,\ldots,expr\,)$.

The second form provides a way of accessing methods of parent classes that have been hidden by overridings in child classes. Instead of using the class of `this` to determine the method, the method of the current superclass is used. For example, if A overrides method f inherited (perhaps indirectly) from B, then `super.`f`()` inside class A invokes the method f in class B on the `this` object.

7.5 Blocks

An explicit block has the form

`{` *block* `}`

where *block* is an implicit block of the form

   $expr\,;\ \ldots\ expr$

The expressions are evaluated in left-to-right order. The value of a block is the value of the last expression, if any, or the unit value otherwise. The expressions of a block may have any static types. The static type of a block is the static type of the last expression, if any, or `Unit`.

An occasional source of confusion in Cool is the use of semi-colons. Semi-colons are used to separate expressions and local declarations in a block and to terminate features, but are not used elsewhere in expressions, as seen in Section 11.

7.6 Local Variables

Any of the expressions inside a block (except the last) may be a local variable declaration of the form:

`var` *id* `:` *type* `=` *expr*`;`

The expression is an *initialization*. When a local declaration is evaluated, the *expr* is evaluated and the result is bound to the *id*.

The binding of *id* is visible later in the block. A local variable may not "shadow" any other declaration (attribute, formal, local or branch) currently in scope.

7.7 Conditionals

A conditional has the form

`if (`*expr*`)` *expr* `else` *expr*

The semantics of conditionals is standard. The predicate is evaluated first. If the predicate is `true`, then the second expression is evaluated. If the predicate is `false`, then the third expression is evaluated. The value of the conditional is the value of the evaluated branch. Despite their close syntactic resemblance to C++'s `if` statements, Cool's `if` expressions function more like C++'s `?:` expressions because they compute values. In particular, the "`else`" part is required in Cool.

Predicates must have static type `Boolean`. The branches may have any static types. To specify the static type of the conditional, we define an operation ∨ (pronounced "join" or "lub" for "least upper bound") on types as follows. Let A, B be types. The *least type* of a set of types means the least element with respect to the conformance relation $\leq$.

$$A \vee B \;\; = \;\; \text{the least type C such that A} \leq \text{C and B} \leq \text{C}$$

Let T, and F be the static types of the branches of the conditional without `elsif`'s. Then the static type of the conditional is $T \vee F$.

7.8 Loops

A loop has the form

```
while (expr) expr
```

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

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

7.9 Pattern Matching

A pattern match expression introduces an expression and gives multiple tests that depend on the exact class of the expression:

```
expr match {
    ( case test => block )+
}
```

Each test has one of the two forms below:

```
    id : class
    null
```

(The type must be a declared class.) Pattern matching expressions provide runtime type tests on objects. First, *expr* is evaluated and its dynamic type C noted. If *expr* evaluates to `null`, the `null` branch is chosen, if any. Otherwise, the first branch *classk* such that $C \leq classk$ is selected. The identifier *idk* is bound to the value of *expr* and the expression *blockk* is evaluated. The result of the `case` is the value of *blockk*. If no branch can be selected for evaluation (for instance, if the expression is null and there is no `null` branch), a run-time error is generated. Every pattern matching expression must have at least one `case` branch.

Every case branch must be *possible*, that is, the branch must have a non-overlapping intersection with the type of the expression being cased. Furthermore, it must not be subsumed by a previous branch (for example, if that previous branches used the same class or a superclass of it).

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

Pattern matching expressions have no special construct for a "default" or "otherwise" branch. The effect is approximated by including a branch

```
case x : Any => ...
```

because every type is $\leq$ to `Any`, except that this does not handle `null`.

Pattern matching 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 A. However, the programmer may know that, in fact, the dynamic type of e is always B for some $B \leq A$. This information can be captured using a case expression:

e `match { case x : ` B ` => ...  }`

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

Case (branch) variables *may* "shadow" an existing declaration of the same name, in which case the latter binding is not visible in the block.

7.10   New

A `new` expression has the form

`new` *type* `(`*expr*`,...,`*expr*`)`

The value is a fresh object of the appropriate class. The static type is *type*, which must name a class. The constructor for the type is called and passed the given arguments. The class must have the same number of formals as expressions are given here and the types must match (as in a dispatch). Internally, the constructor call is seen as a dispatch on a newly created object. You may not create an instance of `Any` or `Symbol`, or of a value type in this way.

7.11   Arithmetic and Comparison Operations

Cool has four binary arithmetic operations: `+, -, *, /`. The syntax is e_1 *op* e_2. To evaluate such an expression first e_1 is evaluated and then e_2. 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 three comparison operations: `<, <=, ==`. For `<` and `<=` the rules are exactly the same as for the binary arithmetic operations, except that the result is a `Boolean`.

The comparison `==` is *syntactic sugar*: e_1 `==` e_2 is parsed into an internal form equivalent to e_1`.equals(`e_2`)` and thus will crash at runtime if e_1 is null.

Finally, there is one arithmetic and one logical unary operator. The expression `-`*expr* is the negation of *expr*. The expression *expr* must have static type `Int` and the entire expression has static type `Int`. The expression `!`*expr* is the boolean complement of *expr*. The expression *expr* must have static type `Boolean` and the entire expression has static type `Boolean`.

8   Basic Classes

There are several predefined classes that the compiler has special information for: `Any`, `IO`, `Unit`, `Int`, `Boolean`, `String`, `Symbol`, and `ArrayAny`. The file `$CLASSHOME/lib/basic.cool` defines and documents each of these classes. Please read this file for more details.

9   Main Class

Every program must have a class `Main` with zero parameters. A program is executed by evaluating `new 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 13), and operational semantics (Section 14).

10   Lexical Structure

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

10.1   Integers, Identifiers, and Special Notation

Integer literals are non-empty strings of digits 0-9 not starting with zero, except that `0` is a valid integer literal. 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. The special syntactic symbols (e.g., parentheses, equal signs, etc.) are shown as they occur in the syntax diagrams.

10.2   Strings

There are two forms for string literals. The simple version of string literals are enclosed in double quotes `"..."`. Within a string literal, a sequence '\c' is illegal unless it is one of the following:

\0	NUL
\b	backspace
\t	tab
\n	newline
\r	return
\f	form feed
\"	double quote
\\	backslash

A newline character may not appear in the simple form of a string literal (even if escaped):

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

The other form of string literals starts with *three* double quotes and continues until ended by three double quotes (again). Any characters, including backslashes and newlines are legal. The only thing forbidden is three double quotes. Thus the following represents a string literal:

```
"""This starts a string literal
that continues on for several lines
even though it includes "'s and \'s and newline characters
in a "wild" profu\sion\\ of normally i\\egal t"ings.\"""
```

The string contains literally everything inside of it.

10.3   Comments

There is two forms for comments in Cool. Any characters after two slashes `//` and before the next newline (or EOF, if there is no next newline) are ignored. Also, any characters excepting the sequence `*/` may be enclosed in C-style comments: `/*`...`*/`.

10.4   Keywords

The keywords of Cool are **case class def else extends false if match native new null override super this true var while** and must be written in lowercase. The keyword **native** may only be used in the special `basic.cool` file provided with the compiler. Additionally, the following words are reserved for "Extended Cool" and cannot be used: **abstract catch do final finally for forSome implicit import lazy object package private protected requires return sealed throw trait try type val with yield**. (We call these the "illegal keywords.")
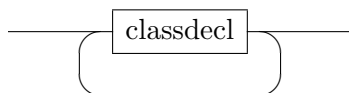
10.5   White Space

White space consists of any sequence of the ASCII characters: HT, NL, CR, and SP. These characters are represented by the following C/C++ character literals respectively: `'\t'`, `'\n'`, `'\r'`, and `' '`.
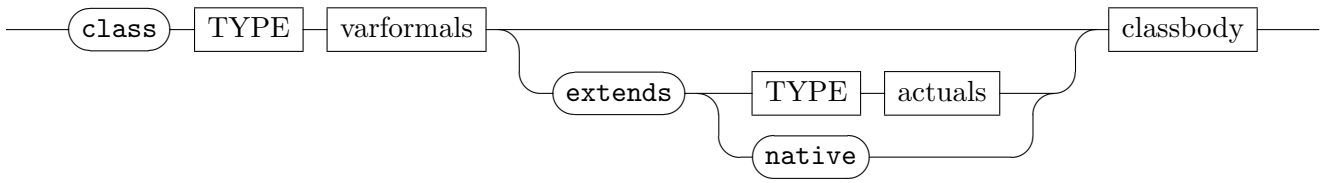
11   Cool Syntax

The grammar is specified here by a sequence of "railroad diagrams" which give grammatical possibilities as paths through a network which do not make any sharp turns (as railway tracks don't). Literal tokens are written inside of ovals; subnetworks are given by a rectangular box naming the nonterminal (lowercase) or terminal (UPPERCASE).
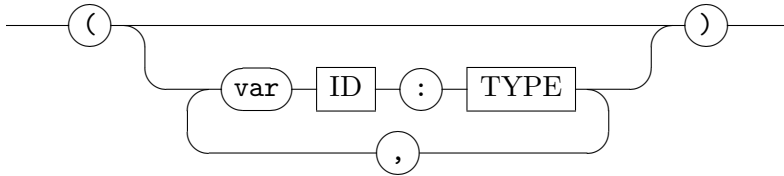
The first diagrams specify the form of a Cool program:
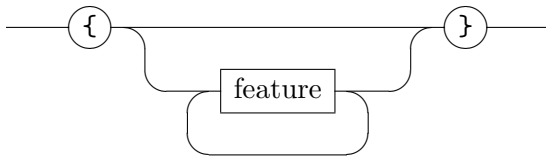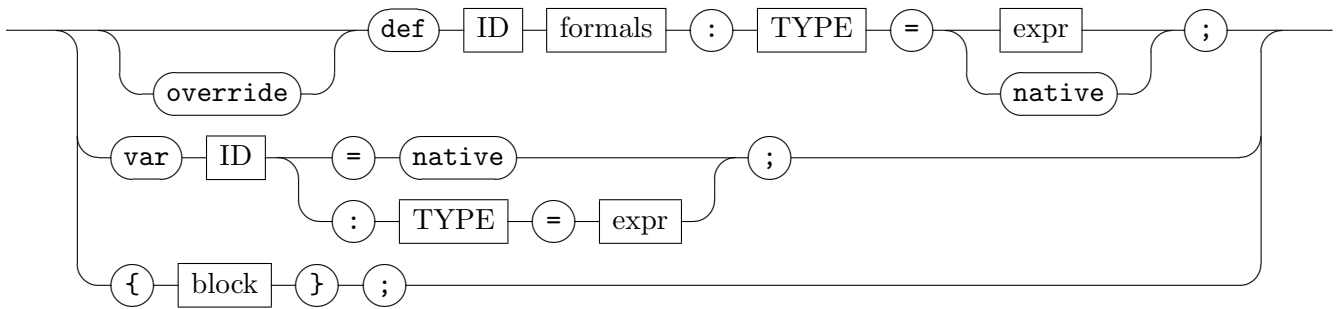
program

classdecl

(class) — TYPE — varformals — classbody
  (extends) — TYPE — actuals
  (native)

varformals

(( )
  (var) — ID — (:) — TYPE
  ,
( ))

classbody

({)
  feature
(})

feature

(override) — (def) — ID — formals — (:) — TYPE — (=) — expr — (;)
  (native)
(var) — ID — (=) — (native) — (;)
  (:) — TYPE — (=) — expr
({) — block — (}) — (;)

formals

(( )
  ID — (:) — TYPE
  ,
( ))

actuals

(( )
  expr
  ,
( ))

13
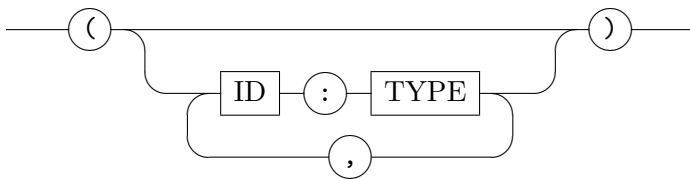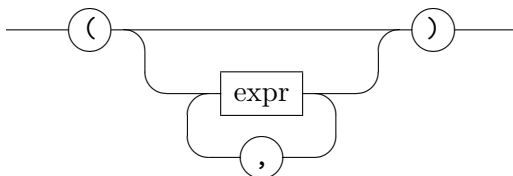
block
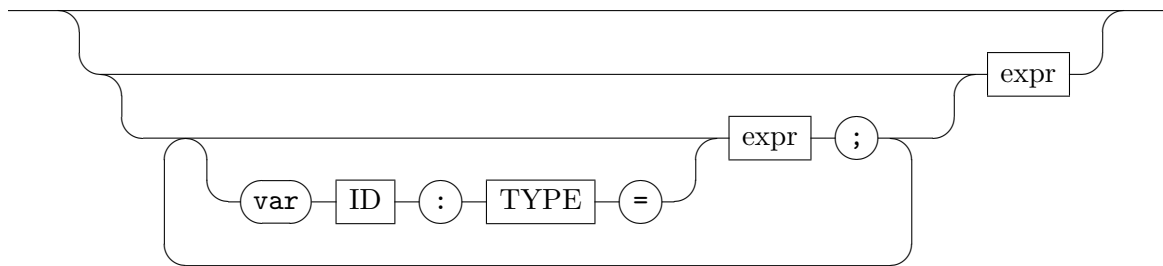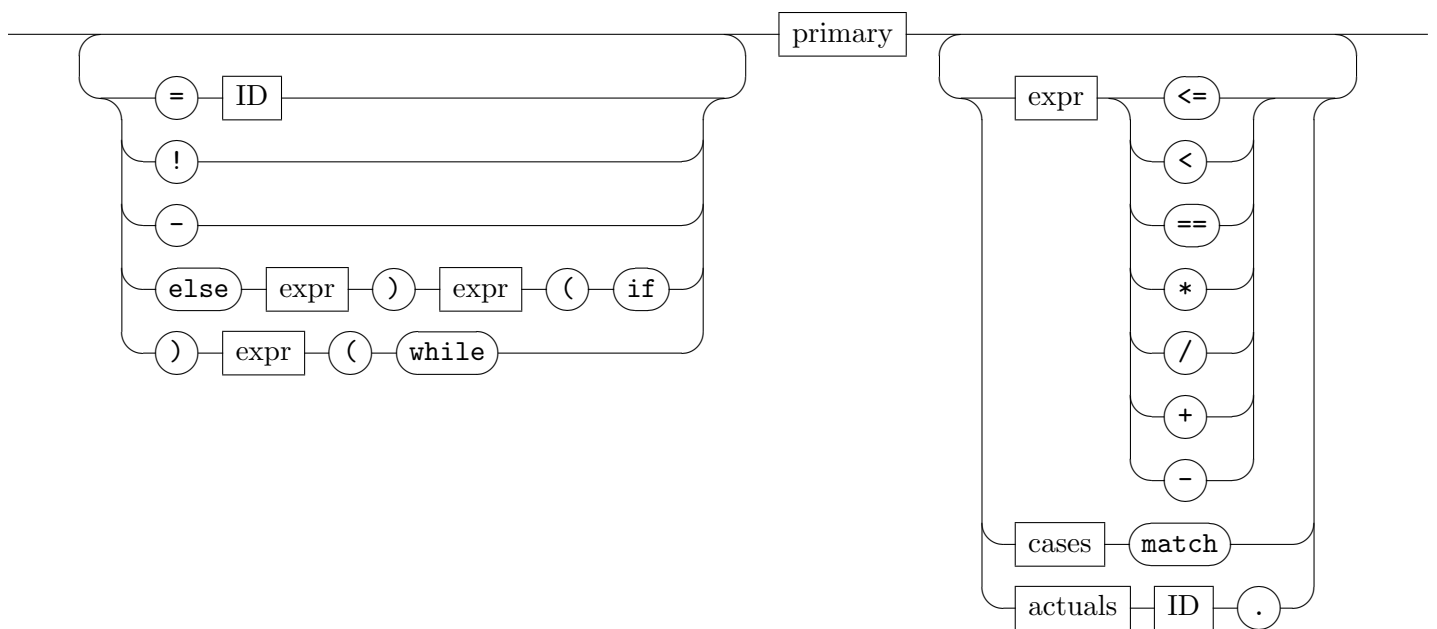
expr

The parentheses for an "if" expression may look backwards, but if one looks carefully, they see that the train will be traveling from right to left and thus will pass the open (left) parenthesis before the closing (right) parenthesis. The same is true for "while."

primary
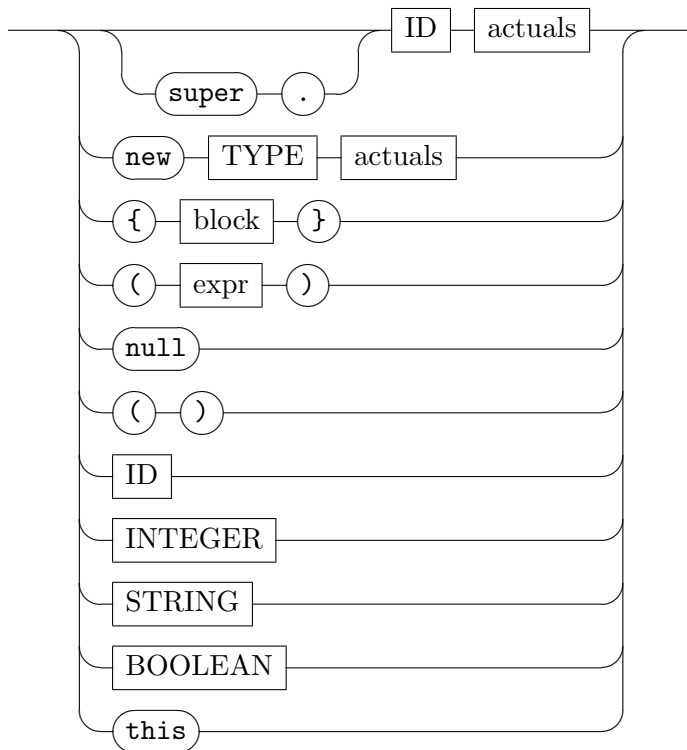
cases
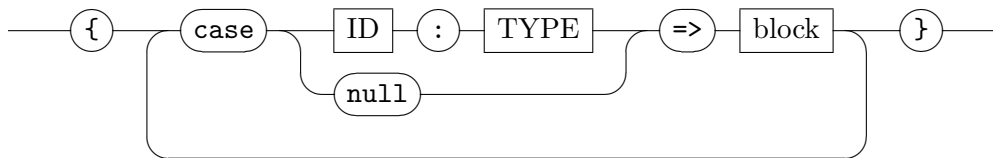
11.1 Precedence

The precedence of infix binary and prefix unary operations, from highest to lowest, is given by the following table (where `-#` means unary minus):

```
.
! -#
* /
+ -
==
<=  <
match
if while
=
```

All binary operations are left-associative.

12 Intermediate Form

The intermediate form of Cool is simpler than its surface syntax. In particular, all implicit blocks are converted into explicit blocks.

We also convert constructors into regular methods, and change `new` to a simpler form: `new` C creates an instance of class C with its attributes initialized to default values.

In theory, the following transformations are applied successively until none can be applied. As it happens this system of rewrite rules is *confluent* which means that the same final result is achieved regardless of the order in which rules are applied. In the rules, e is a meta-variable referring to some expression and E is a meta-variable referring to one or more expressions separated by semicolons:

$$\begin{aligned}
\texttt{case null => } E &\implies \texttt{case null:Null => } E \\
\texttt{case } i\texttt{:}T \texttt{ => } E &\implies \texttt{case } i\texttt{:}T \texttt{ => \{ } E \texttt{ \}} \quad |E| > 1 \\
\texttt{\{ } e \texttt{ \}} &\implies e \quad \text{optionally} \\
\texttt{\{ \}} &\implies \texttt{()} \quad \text{optionally}
\end{aligned}$$

The following rules are also applied at the same time. Here A refers to a comma-separated list of actuals, B refers to a block, F refers to a comma-separated list of formals, X refers to a list of (unprocessed) Cool 2020 features, where Y refers to a list of cleaned up internal features, C refers to the class, and C' refers to its superclass:

$$\begin{aligned}
\texttt{class } C(F) \texttt{ \{ } X \texttt{ \}} &\implies \texttt{class } C(F) \texttt{ extends Any() \{ } X \texttt{ \}} \\
\texttt{class } C(F) \texttt{ extends } C'(A) \texttt{ \{ } X \texttt{ \}} &\implies \texttt{class } C(F) \texttt{ extends } C'(A) \texttt{ \{ } X \texttt{ \{ this \}\}} \\
\texttt{\{ } B_1 \texttt{ \}; \{ } B_2 \texttt{ \}} &\implies \texttt{\{ } B_1; B_2 \texttt{ \}} \\
\texttt{var } i\texttt{:}T \texttt{ = } E\texttt{; \{ } B \texttt{ \}} &\implies \texttt{\{ } i \texttt{ = } E\texttt{; } B \texttt{ \} var } i\texttt{:}T\texttt{;} \\
\texttt{def } i(A) \texttt{ : } T' \texttt{ = } E'\texttt{; \{ } B \texttt{ \}} &\implies \texttt{\{ } B \texttt{ \} def } i(A) \texttt{ : } T' \texttt{ = } E'\texttt{;} \\
\texttt{class } C(F) \texttt{ extends } C'(A) \texttt{ \{\{ } B \texttt{ \} } Y \texttt{ \}} &\implies \begin{array}{l}\texttt{class } C \texttt{ extends } C' \texttt{ \{} \\ \quad \texttt{def } C(\bar{F})\texttt{:}C \texttt{ = \{ this.}C'(A)\texttt{; } f \texttt{ = } \bar{f}\texttt{; ...; } B \texttt{ \};} \\ F\texttt{; } Y \texttt{ \}}\end{array} \\
\texttt{new } C(A) &\implies \texttt{(new } C\texttt{).}C(A)
\end{aligned}$$

In practice, these transformation rules are implemented by building the intermediate form directly in the parser.

13 Type Rules

This section formally defines the type rules of Cool. The type rules define the type of every (intermediate) 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 13.1. Section 13.3 gives the type rules.

13.1 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 e_2 has type X, then the expression `{ `e_1`; `e_2` }` 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 two parts: a method environment M, and an object environment O. 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 (or the constructor name, since we are using the intermediate form), 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 there may be a method and an object identifier of the same name.

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

```
var c : Int = 33;
...
```

The local declaration introduces a new variable `c` with type `Int`. Let O be the object component of the type environment for the block. Then the rest of the block is type checked in the object type environment

$$O[\texttt{c} : \texttt{Int}]$$

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

$$\begin{aligned} O[v : T](v) &= T \\ O[v : T](v') &= O(v') \ \text{ if } v' \neq v \end{aligned}$$

13.2   Type Operations

We define the $\leq$ and $\vee$ (LUB = least upper bound) operations formally as follows

$$\text{S-Self} \quad C \leq C$$

$$\text{S-Extends} \quad \frac{\texttt{class } C\texttt{(...) extends } C'\texttt{(...)} \quad C' \leq C''}{C \leq C''}$$

$$\text{S-Nothing} \quad \texttt{Nothing} \leq C$$

$$\text{S-Null} \quad \frac{C \notin \{\texttt{Nothing}, \texttt{Boolean}, \texttt{Int}, \texttt{Unit}\}}{\texttt{Null} \leq C}$$

$$\text{G-Compare} \quad \frac{C \leq C'}{C \vee C' = C'}$$

$$\text{G-Commute} \quad \frac{C \vee C' = C''}{C' \vee C = C''}$$

$$\text{G-Extends} \quad \frac{C \not\leq C' \quad \texttt{class } C'\texttt{(...) extends } C''\texttt{(...)} \quad C \vee C'' = D}{C \vee C' = D}$$

13.3 Type Checking Rules

The general form a type checking rule is:

$$\text{Name} \quad \frac{\vdots}{O, M \vdash e : T}$$

The rule should be read: In the type environment for objects O, methods M, 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.

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

$$\text{Var} \quad \frac{O(v) = T}{O, M \vdash v : T}$$

The rule for assignment to a variable is more complex:

$$\text{Assign} \quad \frac{O(v) = T \quad O, M \vdash e_1 : T' \quad T' \leq T}{O, M \vdash v \texttt{ = } e_1 : \texttt{Unit}}$$

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 v in the type environment. The type of the whole expression is $\texttt{Unit}$. This rule omits the extra condition that only attributes and local variables can be assigned.

The type rules for literals are all easy:

$$\text{Unit} \quad O, M \vdash \texttt{()} : \texttt{Unit}$$

$$\text{True} \quad O, M \vdash \texttt{true} : \texttt{Boolean}$$

$$\textsc{False}$$
$$O, M \vdash \texttt{false} : \texttt{Boolean}$$

$$\textsc{Int}$$
$$i \text{ is an integer literal}$$
$$\overline{O, M \vdash i : \texttt{Int}}$$

$$\textsc{String}$$
$$s \text{ is a string literal}$$
$$\overline{O, M \vdash s : \texttt{String}}$$

We use the special `Null` type for `null`:

$$\textsc{Null}$$
$$O, M \vdash \texttt{null} : \texttt{Null}$$

Dispatch expressions are the most complex to type check.

$$\textsc{Dispatch}$$
$$\frac{O, M \vdash e_1 : T_1 \quad \ldots \quad O, M \vdash e_n : T_n \quad \begin{array}{c} O, M \vdash e_0 : T_0 \\ M(T_0, f) = (T'_1, \ldots, T'_n, T_{n+1}) \end{array} \quad T_i \leq T'_i \quad 1 \leq i \leq n}{O, M \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}}$$

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. This rule implies that the type of the expression must be a class type (not `Null` or `Nothing`).

$$\textsc{SuperDispatch}$$
$$\frac{O, M \vdash e_1 : T_1 \quad \ldots \quad O, M \vdash e_n : T_n \quad \begin{array}{c} O(\texttt{super}) = T_0 \\ M(T_0, f) = (T'_1, \ldots, T'_n, T_{n+1}) \end{array} \quad T_i \leq T'_i \quad 1 \leq i \leq n}{O, M \vdash \texttt{super}.f(e_1, \ldots, e_n) : T_{n+1}}$$

A super dispatch is similar except that we use the type of `super` (see below).

The type checking rule for `new` is simple, since the constructor call has been reduced to a regular dispatch. But we still need to check against uninstantiable classes.

$$\textsc{New}$$
$$\frac{T \text{ is a class.} \quad T \notin \{\texttt{Any}, \texttt{Int}, \texttt{Unit}, \texttt{Boolean}, \texttt{Symbol}\}}{O, M \vdash \texttt{new } T : T}$$

The type checking rules for `if` expressions are straightforward. See Section 7.7 for the definition of the $\vee$ operation.

$$\textsc{If}$$
$$\frac{O, M \vdash e_0 : \texttt{Boolean} \quad O, M \vdash e_i : T_i, \quad (1 \leq i \leq 2)}{O, M \vdash \texttt{if } (\ e_0\ )\ e_1 \texttt{ else } e_2 : T_1 \vee T_2}$$

The condition must be exactly `Boolean` type (not `Nothing`).

19

For blocks, we have special cases for blocks of zero and one expressions respectively, and then rules for when the first expression is a local variable declaration versus an expression evaluation.

$$\text{BLOCK-NONE}$$
$$O, M \vdash \texttt{\{\}}: \quad \texttt{Unit}$$

$$\text{BLOCK-ONE}$$
$$\frac{O, M \vdash e : T}{O, M \vdash \texttt{\{ } e \texttt{ \}} : T}$$

$$\text{BLOCK-EXPR}$$
$$\frac{O, M \vdash e_1 : T_1 \qquad O, M \vdash \texttt{\{b\}} : T_n}{O, M \vdash \texttt{\{ } e_1\texttt{; } b \texttt{ \}} : T_n}$$

$$\text{BLOCK-LOCAL}$$
$$\frac{O(x) \text{ undefined} \qquad O, M \vdash e : T_0 \qquad T_0 \leq T \qquad O[x : T], M \vdash \texttt{\{b\}} : T'}{O, M \vdash \texttt{\{ var } x \texttt{ : } \quad T \texttt{ = } e\texttt{; } b \texttt{ \}} : T'}$$

First, the initialization e is type checked in an environment without a definition for x. Thus, the variable x cannot be used in e. Then the rest of the block b is type checked in the environment O extended with the typing $x : T$.

For pattern matching, recall that an null branch is converted into a branch using the pattern `case null:Null` (NB: it is intentional that the identifier is a keyword, since a normal identifier would be illegal).

$$\text{MATCH}$$
$$\frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O[x_1 : T_1], M \vdash e_1 : T_1' \quad \ldots \quad O[x_n : T_n], M \vdash e_n : T_n' \qquad \forall_{0 < i,j \leq n} T_i \leq_{-\text{S-NULL}} T_j \Rightarrow i \leq j \\ \forall_{o < i \leq n} T_i \leq T_0 \text{ or } T_0 \leq T_i \qquad \forall_{0 < i \leq n} T_i \text{ is a class type or } x_i = \texttt{"null"} \end{array}}{O, M \vdash e_0 \texttt{ match \{case } x_1 : T_1 \texttt{ => } e_1 \ldots \texttt{case } x_n : T_n \texttt{ => } e_n \texttt{ \}} : \bigvee_{1 \leq i \leq n} T_i'}$$

Each branch of a pattern matching expression is type checked in an environment where variable x_i has type T_i. The type of the entire expression is the join of the types of its branches. The three extra side-conditions ensure that

- A more specific case is not shadowed by an earlier case; (Here we use $\leq_{-\text{S-NULL}}$ which is the $\leq$ relation except that the rule S-NULL cannot be used.)

- Each case is possible;

- Each type is a class type except for the special `case null` branch.

The first condition implies that a `null` branch (if any) must come first. This condition is not semantically necessary since `null` will not be matched by *any* normal branch. You are permitted, but not required to relax the condition slightly for null types to simply require $T_i = T_j \Rightarrow i = j$.

$$\text{LOOP}$$
$$\frac{O, M \vdash e_1 : \texttt{Boolean} \qquad O, M \vdash e_2 : T_2}{O, M \vdash \texttt{while (} e_1 \texttt{) } e_2 : \texttt{Unit}}$$

The predicate of a loop must have type `Boolean`; the type of the entire loop is always `Unit`.

The type checking rules for the primitive logical, comparison, and arithmetic operations are easy. None of the operands may be of `Nothing` type.

$$\frac{\text{NOT}}{O, M \vdash e_1 : \texttt{Boolean}}$$
$$\overline{O, M \vdash \texttt{!}e_1 : \texttt{Boolean}}$$

COMPARE
$$\frac{O, M \vdash e_1 : \texttt{Int} \qquad O, M \vdash e_2 : \texttt{Int} \qquad op \in \{\texttt{<}, \texttt{<=}\}}{O, M \vdash e_1 \ op \ e_2 : \texttt{Boolean}}$$

NEG
$$\frac{O, M \vdash e_1 : \texttt{Int}}{O, M \vdash \texttt{-}e_1 : \texttt{Int}}$$

ARITH
$$\frac{O, M \vdash e_1 : \texttt{Int} \qquad O, M \vdash e_2 : \texttt{Int} \qquad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{O, M \vdash e_1 \ op \ e_2 : \texttt{Int}}$$

The final cases are type checking rules for attributes and methods. For a class C with superclass 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

$$O_C(x) = T$$

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 rule for checking attribute definitions ensures that the attribute is not defined in a superclass already:

ATTR
$$\frac{T \neq \texttt{Nothing} \qquad O_C(x) = T \qquad O_{C'}(x) \ \text{undefined} \qquad O_C, M \vdash e : T' \qquad T' \leq T}{O_C, M \vdash \texttt{var} \ x \ \texttt{:} \quad T \ \texttt{=} \ e\texttt{;}}$$

The rule for typing methods checks that the parameters are the same and that the return type conforms to that of any overridden method and also checks the body of the method in an environment where O_C is extended with bindings for the formal parameters, `super` and `this`. The type of the method body must conform to the declared return type.

METHOD
$$\frac{M(C', f) \ \text{undefined} \qquad \begin{array}{c} M(C, f) = (T_1, \ldots, T_n, T_0) \\ O_C[\texttt{this} : C][\texttt{super} : C'][x_1 : T_1] \ldots [x_n : T_n], M \vdash e : T \qquad T \leq T_0 \end{array}}{O_C, M \vdash \texttt{def} \ f(x_1 : T_1, \ldots, x_n : T_n) : T_0 \ \texttt{=} \ e\texttt{;}}$$

METHOD-OVERRIDE
$$\frac{\begin{array}{c} M(C, f) = (T_1, \ldots, T_n, T_0) \qquad M(C', f) = (T_1, \ldots, T_n, T_0') \\ T_0 \leq T_0' \qquad O_C[\texttt{this} : C][\texttt{super} : C'][x_1 : T_1] \ldots [x_n : T_n], M \vdash e : T \qquad T \leq T_0 \end{array}}{O_C, M \vdash \texttt{override def} \ f(x_1 : T_1, \ldots, x_n : T_n) : T_0 \ \texttt{=} \ e\texttt{;}}$$

If the body of a method is `native`, then it does not need to be checked:

NATIVE
$$O, M \vdash \texttt{native} : T$$

14 Operational Semantics

This section contains a mostly formal presentation of the operational semantics for the (intermediate form of 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 "this" object. These components are described in the next section. Section 14.2 defines the syntax used to refer to Cool objects, and Section 14.3 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.

14.1 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 13.

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

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*. 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.

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

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'.

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

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:

```
{ var c : Int = 33;
  c * c }
```

When evaluating this expression, we must introduce the new identifier c into the environment before evaluating the rest of the block. 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 local variable declaration.

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.

14.2 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 `Boolean`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:

$$\begin{aligned} &\texttt{Unit}() \\ &\texttt{Int}(5) \\ &\texttt{Boolean}(true) \\ &\texttt{String}(\texttt{length} = 4, \texttt{"Cool"}) \end{aligned}$$

`String`s contain two parts, the length and the actual sequence of ASCII characters.

Arrays are another special case, they have the following form

$$\texttt{Array}(\texttt{length} = n, l_1, l_2, \ldots, l_n)$$

The first value is an (immutable) integer, the rest of the values are locations that can be updated.

14.3 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() {
    var s : String = "Hello";
    def g (y:String) : String = y.concat(s);
    def f (x:Int) : Int = x+1;
}


class A() extends B() {
    var a : Int = 42;
    var b : B = new B();
    override def f(x:Int) : Int = x+a;
}
```

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

$$class(\mathtt{A}) = (\mathtt{s} : \mathtt{String} = \mathit{nil}, \ \mathtt{a} : \mathtt{Int} = \mathtt{Integer}(0), \ \mathtt{b} : \mathtt{B} = \mathit{nil})$$

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:

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

Note that every attribute has an initializing expression which is the *default* of its type. The default of $\mathtt{Int}$ is $\mathtt{Integer}(0)$, the default of $\mathtt{Unit}$ is $\mathtt{Unit}()$, the default of $\mathtt{Boolean}$ is $\mathtt{Boolean}(\mathrm{false})$, and the default of any other type is *nil*. 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:

$$
\begin{aligned}
\mathit{implementation}(A, A) &= (A, \mathtt{super.B();\ a{=}42;\ b{=}(new\ B).B();\ this}) \\
\mathit{implementation}(A, f) &= (A, \mathtt{x}, \mathtt{x{+}a}) \\
\mathit{implementation}(A, g) &= (B, \mathtt{y}, \mathtt{y.concat(s)})
\end{aligned}
$$

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

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

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} and that this body was given in class X'.

The *super* mapping is used to compute the superclass of a class, for example:

$$
\begin{aligned}
\mathit{super}(A) &= B \\
\mathit{super}(B) &= \mathtt{Any} \\
\mathit{super}(\mathtt{Any}) &= \mathtt{Any}
\end{aligned}
$$

24

14.4 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:

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

The rule should be read as: In the context in class C, where *this* 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 the current class and a this object *so*. The current class allows us to perform static dispatches. The this object is just the object to which the identifier `this` refers if `this` appears in the expression. We do not place `this` in the environment and store because `this` 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.

$$\text{ASSIGN}$$
$$\frac{C, so, S_1, E \vdash e_1 : v_1, S_2 \qquad E(v) = l_1 \qquad S_3 = S_2[v_1/l_1]}{C, so, S_1, E \vdash v = e_1 : \texttt{Unit}(), S_3}$$

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 result is the unit value. (This rule neglects to prevent assignment to formal parameters or branch identifiers.)

The rules for identifier references, `this`, and literals are straightforward:

$$\text{VAR}$$
$$\frac{E(v) = l \qquad S(l) = v}{C, so, S, E \vdash v : v, S}$$

$$\text{UNIT}$$
$$C, so, S, E \vdash () : \texttt{Unit}(), S$$

$$\text{NULL}$$
$$C, so, S, E \vdash \texttt{null} : nil, S$$

$$\text{THIS}$$
$$C, so, S, E \vdash \texttt{this} : so, S$$

$$\text{TRUE}$$
$$C, so, S, E \vdash \texttt{true} : \texttt{Boolean}(\text{true}), S$$

$$\text{FALSE}$$
$$C, so, S, E \vdash \texttt{false} : \texttt{Boolean}(\text{false}), S$$

$$\text{INT}$$
$$\frac{i \text{ is an integer literal}}{C, so, S, E \vdash i : \texttt{Int}(i), S}$$

$$\text{STRING}$$
$$\frac{s \text{ is a string literal} \qquad l = length(s)}{C, so, S, E \vdash s : \texttt{String}(\texttt{length} = l, s), S}$$

$$\text{DISPATCH}$$
$$\frac{\begin{array}{c} C, so, S_0, E \vdash e_0 : v_0, S_1 \qquad C, so, S_1, E \vdash e_1 : v_1, S_2 \qquad \vdots \qquad C, so, S_n, E \vdash e_n : v_n, S_{n+1} \\ v_0 = X(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \qquad implementation(X, f) = (X', x_1, \ldots, x_n, e_{n+1}) \\ l_{x_i} = newloc(S_{n+1}), \text{ for } i = 1 \ldots n \text{ and each } l_{x_i} \text{ is distinct} \qquad S_{n+2} = S_{n+1}[v_1/l_{x_1}, \ldots, v_n/l_{x_n}] \\ X', v_0, S_{n+2}, [a_1 : l_{a_1}, \ldots, a_m : l_{a_m}, x_1 : l_{x_1}, \ldots, x_n : l_{x_n}] \vdash e_{n+1} : v_{n+1}, S_{n+3} \end{array}}{C, so, S_0, E \vdash e_0.f(e_1, \ldots, e_n) : v_{n+1}, S_{n+3}}$$

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

The dispatch rules do what one would expect. The receiver object (if any) is evaluated and saved. The arguments are evaluated and saved. Then (for DISPATCH), the method is looked up using the dynamic type (static type for STATICDISPATCH) and the method found is run.

A `new` expression in the simplified intermediate language only allocates an object and initializes the attributes:

$$\text{NEW}$$
$$\frac{\begin{array}{c} class(T_0) = (a_1 : T_1 = v_1, \ldots, a_m : T_m = v_m) \\ l_{a_j} = newloc(S_0), \text{ for } j = 1 \ldots m \text{ and each } l_{a_j} \text{ is distinct} \\ v_0 = T_0(a_1 = l_{a_1}, \ldots, a_m = l_{a_m}) \qquad S_1 = S_0[D_{T_1}/l_{a_1}, \ldots, D_{T_m}/l_{a_m}] \end{array}}{C, so, S_0, E \vdash \texttt{new } T_0 : v_1, S_1}$$

$$\text{IF-TRUE}$$
$$\frac{C, so, S_1, E \vdash e_1 : \texttt{Boolean}(\text{true}), S_2 \qquad C, so, S_2, E \vdash e_2 : v_2, S_3}{C, so, S_1, E \vdash \texttt{if (} e_1 \texttt{) } e_2 \texttt{ else } e_3 : v_2, S_3}$$

$$\text{IF-FALSE}$$
$$\frac{C, so, S_1, E \vdash e_1 : \texttt{Boolean}(\text{false}), S_2 \qquad C, so, S_2, E \vdash e_3 : v_3, S_3}{C, so, S_1, E \vdash \texttt{if (} e_1 \texttt{) } e_2 \texttt{ else } e_3 : v_3, S_3}$$

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

$$\text{BLOCK-NONE}$$
$$C, so, S, E \vdash \texttt{\{\}} : \texttt{Unit}(), S$$

$$\text{BLOCK-ONE}$$
$$\frac{C, so, S_0, E \vdash e_1 : v_1, S_1}{C, so, S_0, E \vdash \texttt{\{ } e_1 \texttt{ \}} : v_1, S_1}$$

$$\text{BLOCK-EXPR}$$
$$\frac{C, so, S_0, E \vdash e_1 : v_1, S_1 \qquad C, so, S_1, E \vdash \texttt{\{b\}} : v_n, S_n}{C, so, S_0, E \vdash \texttt{\{ } e_1\texttt{; } b \texttt{ \}} : v, S_n}$$

$$\text{BLOCK-VAR}$$
$$\frac{\begin{array}{c} C, so, S_1, E \vdash e_1 : v_1, S_2 \\ l_1 = newloc(S_2) \qquad S_3 = S_2[v_1/l_1] \qquad E' = E[l_1/v] \qquad C, so, S_3, E' \vdash \texttt{\{ } b_2 \texttt{ \}} : v_2, S_4 \end{array}}{C, so, S_1, E \vdash \texttt{\{ var } v \texttt{ : } T \texttt{ = } e_1\texttt{; } b_2 \texttt{ \}} : v_2, S_4}$$

With a local, we evaluate any initialization code, assigns the result to the variable at a fresh location, and then evaluate the rest of the block.

$$\text{CASE-NOT-NULL}$$
$$\frac{\begin{array}{c} C, so, S_1, E \vdash e_0 : v_0, S_2 \qquad v_0 = X(\dots) \qquad T_i = \text{first } T_i \text{ where } X \le T_i \\ l_0 = newloc(S_2) \qquad S_3 = S_2[v_0/l_0] \qquad E' = E[l_0/v_i] \qquad C, so, S_3, E' \vdash e_i : v_1, S_4 \end{array}}{C, so, S_1, E \vdash e_0 \texttt{ match \{ case } v_1 : T_1 \texttt{ => } e_1 \dots \texttt{ case } v_n : T_n \texttt{ => } e_n \texttt{ \}} : v_1, S_4}$$

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.

If the expression is null, this rule does not apply. We have a different rule for the case that the expression is null:

$$\text{CASE-NULL}$$
$$\frac{C, so, S_1, E \vdash e_0 : v_0, S_2 \qquad v_0 = nil \qquad C, so, S_2, E \vdash e_i : v_1, S_3}{C, so, S_1, E \vdash e_0 \texttt{ match \{ case } v_1 : T_1 \texttt{ => } e_1 \dots \texttt{ case null => } e_i \dots \texttt{ case } v_n : T_n \texttt{ => } e_n \texttt{ \}} : v_1, S_3}$$

$$\text{LOOP-TRUE}$$
$$\frac{\begin{array}{c} C, so, S_1, E \vdash e_1 : \texttt{Boolean}(\text{true}), S_2 \\ C, so, S_2, E \vdash e_2 : v_2, S_3 \qquad C, so, S_3, E \vdash \texttt{while (} e_1 \texttt{) } e_2 : \texttt{Unit}(), S_4 \end{array}}{C, so, S_1, E \vdash \texttt{while (} e_1 \texttt{) } e_2 : \texttt{Unit}(), S_4}$$

$$\text{LOOP-FALSE}$$
$$\frac{C, so, S_1, E \vdash e_1 : \texttt{Boolean}(\text{false}), S_2}{C, so, S_1, E \vdash \texttt{while (} e_1 \texttt{) } e_2 : \texttt{Unit}(), S_2}$$

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

$$\text{NOT}$$
$$\frac{C, so, S_1, E \vdash e_1 : \texttt{Boolean}(b), S_2 \qquad v_1 = \texttt{Boolean}(\neg b)}{C, so, S_1, E \vdash \; ! \quad e_1 : v_1, S_2}$$

$$\text{COMP}$$
$$\frac{C, so, S_1, E \vdash e_1 : \texttt{Int}(i_1), S_2 \\ C, so, S_2, E \vdash e_2 : \texttt{Int}(i_2), S_3 \qquad op \in \{\leq, <\} \qquad v_1 = \begin{cases} \texttt{Boolean(true)}, & \text{if } i_1 \; op \; i_2 \\ \texttt{Boolean(false)}, & \text{otherwise} \end{cases}}{C, so, S_1, E \vdash e_1 \; op \; e_2 : v_1, S_3}$$

$$\text{NEG}$$
$$\frac{C, so, S_1, E \vdash e_1 : \texttt{Int}(i_1), S_2 \qquad v_1 = \texttt{Int}(-i_1)}{C, so, S_1, E \vdash -e_1 : v_1, S_2}$$

$$\text{ARITH}$$
$$\frac{C, so, S_1, E \vdash e_1 : \texttt{Int}(i_1), S_2 \\ C, so, S_2, E \vdash e_2 : \texttt{Int}(i_2), S_3 \qquad op \in \{+, -, *, /\} \qquad v_1 = \texttt{Int}(i_1 \; op \; i_2)}{C, so, S_1, E \vdash e_1 \; op \; e_2 : v_1, S_3}$$

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

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 `null`.

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

3. Division by zero.

4. index/size out of range for string or array

5. Heap overflow.

Finally, the rules given above do not explain the execution behavior for dispatches to native methods. English descriptions of these methods are given in `basic.cool`.

15   Acknowledgments