The Cool Reference Manual

Introduction

This manual describes the programming language Cool: the $\mathcal{C}$lassroom $\mathcal{O}$bject-$\mathcal{O}$riented $\mathcal{L}$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 $\mathit{classes}$. A class encapsulates the variables and procedures of a data type. Instances of a class are $\mathit{objects}$. In Cool, classes and types are identified. That is, every class defines a type. Classes permit programmers to define new types and associated procedures (or $\mathit{methods}$) specific to those types. Inheritance allows new types to extend the behavior of existing types.

Cool is an $\mathit{expression}$ language. Most Cool constructs are expressions, and every expression has a value and a type. Cool is $\mathit{type\ safe}$: 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.

Getting Started

Cool source files have extension .cl. The programming projects will also define other file formats related to Cool but they are not officially part of the language specification.

You can obtain the Cool interpreter cool from the course website. To interpret (i.e., run) a Cool program:

```
cool file.cl
```

This official version is often called the *reference implementation* and you are encouraged to use it as a point of comparison when you are designing and testing parts of the course project. The reference Cool interpreter has been specifically structured so that you can run the various stages (i.e., lexing, parsing, type-checking and interpreting) independently. This power is useful for PA2 through PA5.

The Cool interpreter has an number of command-line options:

--lex
: This option causes Cool to stop after lexing and produce a new file, file.cl–lex, which contains the lexed tokens in a simple interchange format. Handy for PA2.

--unlex
: This option causes Cool to undo lexing and produce file.cl2 (a Cool source file) from the lexed tokens. This is usually used as a debugging aid for PA2 by feeding Cool a file.cl–lex file produced by *your lexer* from file.cl and then comparing file.cl2 to the original file.cl.

--parse
: This option causes Cool to stop after parsing and produce a new file, file.cl–ast, which contains the abstract syntax tree in a simple interchange format. Handy for PA3.

--unparse
: This option causes Cool to undo parsing and produce file.cl3 (a Cool source file) from the abstract syntax tree. This is usually used as a debugging aid for PA3 by feeding Cool a file.cl–ast file produced by your parser from file.cl and then comparing file.cl3 to the original file.cl.

--type
: This option causes Cool to stop after type checking and semantic analysis and produce a new file, file.cl–type, which contains the class mapping and implementation mapping in a simple interchange format. Handy for PA4.

--class-map
: This option causes Cool to stop after type checking and semantic analysis and produce a new file, file.cl–type, which contains the class mapping only in a simple interchange format. Handy for the PA4 Checkpoint (WA4).

--imp-map
: This option causes Cool to stop after type checking and semantic analysis and produce a new file, file.cl–type, which contains the implementation mapping only in a simple interchange format. Handy for the rest of PA4.

--parent-map
: This option causes Cool to stop after type checking and semantic analysis and produce a new file, file.cl–type, which contains the parent mapping only in a simple interchange format. Handy for the rest of PA4.

--out *newname*
: Causes Cool to produce *newname.cl* (or .cl–ast, or whatever) instead of file.cl.

You may encounter other University uses of Cool on the web that mention programs such as coolc and spim. Those tool are used for a course on compilers; this is a course on interpreters. We will not use coolc or spim. In addition, we use a slightly different version of the Cool language specification, so comparing results against external tools may not be

helpful.

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.

- Features
- Inheritance

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 $\mathrm{list.cl}$ 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.(2) 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.

Inheritance

If a class definition has the form

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

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

There is a distinguished class Object. If a class definition does not specify a parent class, then the class inherits from Object by default. A class may inherit only from a single class; this is aptly called "single inheritance."(3) 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.

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 `C` $\leq$ `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 `A`, `C`, and `P` be types.

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

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

- SELF_TYPE
- Type Checking

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

```
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 $(\text{newSally}).\text{copy}()$ 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 $\texttt{SELF\_TYPE}_{\text{C}}$. 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 $\texttt{SELF\_TYPE}_{\text{X}} \leq \texttt{SELF\_TYPE}_{\text{X}}$. There is also a special conformance rule for SELF_TYPE:

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

Finally, SELF_TYPE may be used in the following places: $\text{new}\ \text{SELF\_TYPE}$, 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.

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.

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.

- Void

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.

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.

Expressions

Expressions are the largest syntactic category in Cool.

- Constants
- Identifiers
- Assignment
- Dispatch
- Conditionals
- Loops
- Blocks
- Let
- Case
- New
- Isvoid
- Arithmetic and Comparison Operations

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 $\text{“This is a string.”}$ 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.

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.

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 $>$.

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 $\text{self}.<\text{id}>(<\text{expr}>,\ldots,<\text{expr}>)$.

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, $\text{e@B}.\text{f}()$ 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 "@".

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 $\text{A}, \text{B}, \text{D}$ 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 $\leq$.

$$\begin{aligned}
\texttt{A} \sqcup \texttt{B} &= \texttt{the least type C such that A} \leq \texttt{C and B} \leq \texttt{C} \\
\texttt{A} \sqcup \texttt{A} &= \texttt{A} \qquad \texttt{(idempotent)} \\
\texttt{A} \sqcup \texttt{B} &= \texttt{B} \sqcup \texttt{A} \qquad \texttt{(commutative)} \\
\texttt{SELF_TYPE}_{\texttt{D}} \sqcup \texttt{A} &= \texttt{D} \sqcup \texttt{A}
\end{aligned}$$

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

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.

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.

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 $< \text{expr1} >$ is evaluated and the result bound to $< \text{id1} >$. Then $< \text{expr2} >$ is evaluated and the result bound to $< \text{id2} >$, and so on, until all of the variables in the let are initialized. (If the initialization of $< \text{idk} >$ is omitted, the default initialization of type $< \text{typek} >$ is used.) Next the body of the let is evaluated. The value of the let is the value of the body.

The let identifiers $< \text{id1} >, \ldots, < \text{idn} >$ are visible in the body of the let. Furthermore, identifiers $< \text{id1} >, \ldots, < \text{idk} >$ are visible in the initialization of $< \text{idm} >$ for any $\text{m} > \text{k}$.

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 $< \text{expr} >$ of a let extends as far (encompasses as many tokens) as the grammar allows.

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 $< \text{typek} >$ such that $\text{C} \leq < \text{typek} >$ is selected. The identifier $< \text{idk} >$ is bound to the value of $< \text{expr0} >$ and the expression $< \text{exprk} >$ is evaluated. The result of the case is the value of $< \text{exprk} >$. 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 $< \text{expri} >$. The static type of a case expression is $\bigsqcup_{1 \leq \text{i} \leq \text{n}} \text{T}_\text{i}$. 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`.

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 $< \text{type} >$.

Isvoid

The expression

```
isvoid expr
```

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

Arithmetic and Comparison Operations

Binary Arithmetic

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 entire arithmetic expression is also Int. Cool has only integer division.

Binary Relations

Cool has three comparison operations: $<$, $<=$, $=$. These comparisons may be applied to subexpressions of any types, subject to the following rules:

- expr1 is an Int if and only if expr2 is an Int
- expr1 is a String if and only if expr2 is a String
- expr1 is a Bool if and only if expr2 is a Bool
- Otherwise, expr1 may be of any type (including SELF_TYPE) and expr2 may be of any (possibly different) type (including SELF_TYPE).

In all cases, the result of the comparison is a Bool. See the type checking rules for more information.

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.

On non-basic objects, equality is decided via pointer equality (i.e., whether the memory addresses of the objects are the same). Equality is defined for void: two void values are equal and a void value is never equal to a non-void value. See the operational semantics rules for more informaiton.

Unary Expressions

Finally, there is one unary arithmetic and one unary logical operator.

- The expression $\sim \ < \text{expr} >$ is the integer complement of $< \text{expr} >$. The subexpression $< \text{expr} >$ must have static type Int and the entire expression has static type Int.
- The expression $\text{not} < \text{expr} >$ is the boolean complement of $< \text{expr} >$. The subexpression $< \text{expr} >$ must have static type Bool and the entire expression has static type Bool.

Basic Classes

- Object
- IO
- Int
- String
- Bool

Object

The Object class is the root of the inheritance graph. Even the other basic classes (e.g., IO and Int) inherit from Object (and thus inherit the three methods listed below). It is an error to redefine Object. Methods with the following declarations are defined:

```
abort() : Object
type_name() : String
copy() : SELF_TYPE
```

The method abort flushes all output and then halts program execution with the error message "abort\n".

The method type_name returns a string with the name of the (run-time, dynamic) class of the object.

The method copy produces a *shallow* copy of the object.(4)

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
```

The methods out_string and out_int print their argument, flush the standard output, and return their self parameter.

The interpreter or compiler changes every \t to a tab and every \n to a newline in the argument x to out_string before emitting the resulting string. Note that this is different from normal escape sequence handling, where \n would be a single character stored in the string. In Cool, it is two characters, but out_string prints a newline instead of \n.

The method in_string reads a string from the standard input, up to but not including a newline character or the end of file. The newline character is consumed but is not made part of the returned string. If an error occurs then in_string returns "", the string of length 0. Note that while literal lexical string constants are limited to size 1024, strings generated by in_string (or String.concat, etc.) can be of arbitrary size. There is no special processing of the two-character sequences \t or \n (or, indeed *anything*) during in_string. Errors include:

- no input found before end of file
- string read in contains NUL, the character with ASCII value 0 (in which case the entire string is rejected, even if there are valid characters around the NUL)

The method in_int reads a single possibly-signed integer, which may be preceded by whitespace. Any characters following the integer, up to and including the next newline, are discarded by in_int. If an error occurs then in_int returns 0. Errors include:

- no input found before end of file
- malformed input (i.e., first thing after whitespace is not a possibly-signed integer)

- integer read in is $<$ -2147483648
- integer read in is $>$ +2147483647

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.

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.

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. Substring errors are always reported as taking place on line 0.

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

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.

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 may be defined in class Main or it may be inherited from another class. A program is executed by evaluating (newMain).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).

Lexical Structure

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

Integers, Identifiers, and Special Notation

Integers are non-empty strings of digits 0-9. It is a lexer error if a literal integer constant is too big to be represented as a 32-bit signed integer. 32-bit signed integers range from -2,147,483,648 to +2,147,483,647. Cool integer constants are always non-negative, so valid integer constants range from 0 to 2,147,483,647.

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. Identifiers are case sensitive.

self and **SELF_TYPE** are treated specially by Cool but are not treated as keywords. **self** should be reported by the lexer as an identifier and **SELF_TYPE** should be reported by the lexer as a type. Both are case sensitive.

The special syntactic symbols (e.g., parentheses, assignment operator, etc.) are given in Figure 1.

Strings

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

```
\t tab
\n newline
```

The two-character sequences $\backslash n$ and $\backslash t$ are called $escape\ sequences$. Other escape sequences like $\backslash r$ (carriage return) are not part of Cool. These two special escape sequences should not be interpreted or transformed by the lexer; they are handled by the IO module and the run-time system.

A newline character may not appear in a string:

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

A string may contain embedded double quotes, so long as they are escaped. The following is a valid Cool string:

```
"David St. Hubbins said, \"It's such a fine line between stupid, and clever.\""
```

Note that Cool's interpretation of \ ” may not be what you are expecting. The two-character sequence \ ” (which is not an escape sequence) does not become ” in any sense. Instead, it stays \ ”. This is different from most other languages, but simplifies lexing and interpreting. Example:

```
class Main inherits IO {
        main() : Object {
                out_string("She said, \"Hello.\"\n")
        } ;
} ;
```

- Output: $\text{She said, } \backslash\text{“Hello.} \backslash\text{”}$

A string may not contain EOF; strings cannot cross file boundaries. A string may not contain NUL, the character with ASCII value 0. The lexer must reject source text that contains malformed strings.

A string may contain the two-character sequence $\backslash 0$ (backslash zero). However, that sequence does not have any special meaning -- it just yields a backslash followed by a zero inside the string.

The single character with converted integer value zero (the NUL) is not allowed. Any other character may be included in a string.

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 but may not contain EOF.

Comments cannot cross file boundaries.

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. Thus **FALse** is not a keyword but a type identifier.

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).

Cool Syntax

$$
\begin{aligned}
\mathit{program} &::= [\![\mathit{class};]\!]^+ \\
\mathit{class} &::= \texttt{class TYPE } [\texttt{inherits TYPE}]\{\ [\![\mathit{feature};]\!]^*\ \} \\
\mathit{feature} &::= \texttt{ID}(\ [\ \mathit{formal}[\![,\mathit{formal}]\!]^*\ ]) : \texttt{TYPE}\ \{\ \mathit{expr}\ \} \\
&\mid\ \texttt{ID : TYPE}\ [\ \leftarrow \mathit{expr}\ ] \\
\mathit{formal} &::= \texttt{ID : TYPE} \\
\mathit{expr} &::= \texttt{ID} \leftarrow \mathit{expr} \\
&\mid\ \mathit{expr}[\texttt{@TYPE}].\ \texttt{ID}(\ [\ \mathit{expr}\ [\![,\mathit{expr}]\!]^*\ ]\ ) \\
&\mid\ \texttt{ID}(\ [\ \mathit{expr}\ [\![,\mathit{expr}]\!]^*\ ]\ ) \\
&\mid\ \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr}\ \texttt{fi} \\
&\mid\ \texttt{while}\ \mathit{expr}\ \texttt{loop}\ \mathit{expr}\ \texttt{pool} \\
&\mid\ \{\ [\![\mathit{expr};]\!]^+\ \} \\
&\mid\ \texttt{let ID : TYPE}\ [\ \leftarrow \mathit{expr}\ ]\ [\![\texttt{, ID : TYPE}\ [\ \leftarrow \mathit{expr}\ ]]\!]^*\ \texttt{in}\ \mathit{expr} \\
&\mid\ \texttt{case}\ \mathit{expr}\ \texttt{of}\ [\![\texttt{ID : TYPE} \Rightarrow\ \mathit{expr};]\!]^+\ \texttt{esac} \\
&\mid\ \texttt{new TYPE} \\
&\mid\ \texttt{isvoid}\ \mathit{expr} \\
&\mid\ \mathit{expr}\ +\ \mathit{expr} \\
&\mid\ \mathit{expr}\ -\ \mathit{expr} \\
&\mid\ \mathit{expr}\ *\ \mathit{expr} \\
&\mid\ \mathit{expr}\ /\ \mathit{expr} \\
&\mid\ \sim \mathit{expr} \\
&\mid\ \mathit{expr} < \mathit{expr} \\
&\mid\ \mathit{expr} <= \mathit{expr} \\
&\mid\ \mathit{expr} = \mathit{expr} \\
&\mid\ \texttt{not}\ \mathit{expr} \\
&\mid\ (\mathit{expr}) \\
&\mid\ \texttt{ID} \\
&\mid\ \text{integer} \\
&\mid\ \text{string} \\
&\mid\ \texttt{true} \\
&\mid\ \texttt{false}
\end{aligned}
$$

Figure 1 : 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. Items in square brackets $[\ldots]$ are optional. Double brackets $[\![]\!]$ 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).

- Precedence

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.

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.

- Type Environments
- Type Checking Rules

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 $\mathtt{X}$, then the expression $\{\ \mathtt{e_1}\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{e_n}\mathtt{;}\ \}$ has type $\mathtt{X}$.

A complication arises in the case of an expression $\mathtt{v}$, where $\mathtt{v}$ is an object identifier. It is not possible to say what the type of $\mathtt{v}$ is in a strictly bottom-up algorithm; we need to know the type declared for $\mathtt{v}$ 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 $\mathrm{SELF\_TYPE}$.

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:

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

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 "turnstile" ("$\vdash$") 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.

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

The rule for assignment to a variable is more complex:

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

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:

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

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

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

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

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

$$\frac{T' = \begin{cases} \text{SELF\_TYPE}_C & \text{if } T = \text{SELF\_TYPE} \\ T & \text{otherwise} \end{cases}}{O, M, C \vdash new\ T : T'} [\text{New}]$$

Dispatch expressions are the most complex to type check.

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

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

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.

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

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

The let rule has some interesting aspects.

$$\frac{\begin{array}{l} T_0' = \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0' \\ O[T_0'/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{let } x : T_0 \leftarrow e_1 \textit{ in } e_2 : T_2}\,[\text{Let-Init}]$$

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.

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

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

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

is defined to be the same as typing

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

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

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.

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

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

$$\frac{O, M, C \vdash e_1 : T_1}{O, M, C \vdash \text{isvoid } e_1 : \mathit{Bool}} [\text{Isvoid}]$$

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

$$\frac{O, M, C \vdash e_1 : \mathit{Bool}}{O, M, C \vdash \neg e_1 : \mathit{Bool}} [\text{Not}]$$

$$\frac{O, M, C \vdash e_1 : \mathit{Int}}{O, M, C \vdash \sim e_1 : \mathit{Int}} [\text{Neg}]$$

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

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. The cases for $<$ and $<=$ are similar to the rule for equality.

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

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

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

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.

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

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

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.

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

Other Semantic Checks

There are a number of semantic checks applied to Cool programs that are not captured by formal typing rules. For example, a Cool program cannot contain an inheritance cycle. Similarly, a Cool program cannot contain a class that inherits from String. These rules are scattered through the *Cool Reference Manual*.

The order in which these other checks are performed is *not specified*. If a Cool program contains both an inheritance cycle and also a class that inherits from String, the Cool compiler may report whichever error it prefers.

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.

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.

$$E(a) = l_1$$
$$S(l_1) = 55$$

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'.

$$E(a) = l_1$$
$$S' = S[99/l_1]$$

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.

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. The value v has dynamic type X.

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:

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

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

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:

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

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. (5) 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:

$$implementation(A, f) = (x, x + a)$$
$$implementation(A, g) = (y, y.concat(s))$$

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}.

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.

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

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:

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

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

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

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

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

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

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

The tricky thing in a `new` expression is to initialize the attributes in the right order. If an attribute does not have an initializer, *do not* evaluate an assignment expression for it in the final step. Note also that, during initialization, attributes are bound to the default of the appropriate class.

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

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

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.

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

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

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

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

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

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

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`

$$let\ x_1 : T_1 \leftarrow e_1, x_2 : T_2 \leftarrow e_2, \ldots, x_n : T_n \leftarrow e_n\ in\ e$$

is defined to be the same as

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

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

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.

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

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

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:

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

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

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

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

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

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

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. < and <= are handled similarly. The case for integer arguments is simple:

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

... but `String` and `Bool` also admit comparisons. String comparisons are performed using the standard ASCII string ordering (e.g., `" abc "<" xyz "`). For booleans, `false` is defined to be less than `true`. Any other comparison (e.g., a comparison among non-void objects of different types) returns `false`. Note that for some objects this may be unintuitive: if `c` is a `Cat` and `d` is a `Dog` then `c < d` is `false` but `d < c` is also `false`. Note also that comparison is based on the dynamic type of the object, not on the static type of the object.

In addition, the operational rules do not specify what happens in the event of a runtime error. When a runtime error occurs, output is flushed and execution aborts. 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.
   (This error is always reported on line 0, since it occurs inside an "internal" library function.)
6. Heap overflow. *(You do not need to implement this runtime error.)*
7. Stack overflow.

Each outstanding "method invocation" (static or dynamic) and each outstanding "new" object allocation expression counts as a "Cool Activation Record". (Just to be clear, that second clause about "new" is counting currently-resolving constructor calls, not "total objects living in the heap".) A Cool interpreter *must* flag a "stack overflow" runtime error if and only if there are **1000** (one thousand) or more outstanding Cool Activation Records.

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.

Cool Assembly Language

Cool Assembly Language is a simplified RISC-style assembly language that is reminiscent of MIPS Assembly Language crossed with x86 Assembly Language. It also features typing aspects that may remind one of Java Bytecode.

A Cool Assembly Language **program** is a list of **instructions**. Each instruction may be preceded by any number of **labels**. Comments follow the standard Cool conventions. In addition, a semicolon ; functions like a double dash −− in that it marks the rest of that line as a comment. The Cool CPU is a load-store architecture with eight general purpose registers and three special-purpose registers. For simplicity, a machine word can hold either a 32-bit integer value or an entire raw string; regardless, all machine words have size one.

This document assumes that you already have some familiarity with assembly language, registers, and how CPUs operate. We first present a formal grammar and then explain the semantics. Only terms in typewriter font are part of the formal grammar. Text after -- is a comment. We use *italics* for non-terminals.

register ::= r0 -- general purpose register #0, often used as the *accumulator*
register ::= r1 -- general purpose register #1
register ::= r2
register ::= r3
register ::= r4
register ::= r5
register ::= r6
register ::= r7
register ::= sp -- stack pointer register
register ::= fp -- frame pointer register
register ::= ra -- return address pointer

instruction ::= li *register* -- load immediate
instruction ::= mov *register* <- *register* -- register-to-register copy
instruction ::= add *register* <- *register register*
instruction ::= sub *register* <- *register register*
instruction ::= mul *register* <- *register register*
instruction ::= div *register* <- *register register*

instruction ::= jmp *label* -- unconditional branch
instruction ::= bz *register label* -- branch if equal to zero
instruction ::= bnz *register label* -- branch if not zero
instruction ::= beq *register register label* -- branch if equal
instruction ::= blt *register register label* -- branch if less than
instruction ::= ble *register register label* -- branch if less than or equal to
instruction ::= call *label* -- direct function call
instruction ::= call *register* -- register-indirect function call
instruction ::= return -- function return

instruction ::= push *register* -- push a value on the stack
instruction ::= pop *register* -- push a value off the stack
instruction ::= ld *register* <- *register* [*integer*] -- load a value from memory
instruction ::= st *register* [*integer*] <- *register* -- store a value into memory
instruction ::= la *register* <- *label* -- load an address into a register

instruction ::= alloc *register register* -- allocate memory
instruction ::= constant *integer* -- lay out a compile-time constant in memory
instruction ::= constant *raw_string* -- lay out a compile-time constant in memory
instruction ::= constant *label* -- lay out a compile-time constant in memory

instruction ::= syscall *name* -- request a service from the run-time system

instruction ::= debug *register* -- debugging support: print register value
instruction ::= trace -- toggle tracing

That's it, and the last two do not really count. We next describe the interpretation of these instructions in more detail.

- Note that there are only eight general purpose registers available, as with the x86 instruction set. This is a departure from general RISC, but it will give you more of a feel for the real world. Eight is entirely sufficient for a stack-machine style of code generation -- the reference compiler only uses four of them. However, for advanced optimizations such as register allocation, eight is quite small.
- li r1 <- some_int overwrites r1 with the value some_int
- Note that in Cool Assembly Language, the arrows <- are required. They remind you that the destination is always on the left and the operands are always on the right.
- bz r1 label jumps to label if the value of r1 is zero. If not, control passes to the instruction immediately following this one.
- beq r1 r2 label jumps to label if the registers r1 and r2 hold the same value.
- ble r1 r2 label jumps to label if the value of r1 is less than or equal to the value of r2. For integers this is standard. If r1 and r2 hold raw strings, those strings are compared lexicographically.
- call label stores the value of the next instruction (i.e., the value of the current program

counter + 1) in the ra register and then jumps to label.

- call *register* stores the value of the next instruction (i.e., the value of the current program counter + 1) in the ra register and then jumps to address stored in *register*.
- return jumps to the address stored in the ra register.
- Like the x86, the Cool CPU has explicit support for a stack. The stack pointer starts at a very high value and *grows down*, toward smaller numbers, as values are pushed on it. push r1 takes the value of r1 and stores it at the address given by the stack pointer sp and then decrements sp. pop r1 increments sp and then loads the value from the address given by the stack pointer and copies that value into r1.
- ld r1 <- r2 [offset] computes an address by adding offset to the value stored in r2. The contents of that address are loaded and written to r1.
- st r1 [offset] <- r2 computes an address by adding offset to the value stored in r1. The contents of r2 are stored into that address in memory.
- la r1 label stores the address associated with label into r1.
- alloc r1 r2 allocates new contiguous memory and stores a pointer to it in r1. The number of words to be allocated is given in r2. For example, if r2 = 5 and alloc r1 r2 assigns 100 into r1, then 100 through 104 are now valid memory addresses.
- constant whatever lays out whatever in program memory before execution begins.

The system calls available are:

- syscall IO.in_string reads a raw string from the user, allocates one word of memory, stores the raw string value in that memory word, and then stores the address of that memory word in r1. Note that this yields a raw string value and not a Cool String object -- you'll have to do a bit more work to package it up into a full-fledged Cool String object.
- syscall IO.in_int reads an integer from the user and stores that integer value in r1. Note that this yields a raw integer value and not a Cool Int object.
- syscall IO.out_int reads the value in r1 and displays it as an integer to the user. Note that r1 should be a raw integer and not an entire large Cool Int object.
- syscall IO.out_string reads the value in r1, which should be an address that points to a machine word containing a raw string. That raw string value is read from memory and displayed to the user. Note that r1 should be a pointer to a raw string, and not a large Cool String object.
- syscall String.length reads the value in r1, which should be an address that points to a machine word containing a raw string. The length of that string value is computed and stored in r1.
- syscall String.concat reads the values in r1 and r2, both of which should be addresses that point to machine words that contain raw strings. A machine word for a new string is allocated in memory. That new string contains the r1-string concatenated with the r2-string. The register r1 is overwritten so that it contains a pointer to the newly-created raw string.
- syscall String.substr reads the value in r0, which should be an address that points to a machine word containing a raw string, as well as r1 and r2, which are both raw integer values.
    - If r1<0, r2<0, or r1+r2> the length of the raw string, the system call stores 0 in r1.
    - Otherwise, a word is allocated in memory to hold a new raw string. That new raw string holds the substring specified by the three arguments. The address of that new raw string is stored in r1.

- syscall exit terminates execution of the Cool Assembly Language program.

That system calls correspond directly to internal predefined methods on Cool Int and String objects. The key difference is that the system calls work on raw values (i.e., machine-level ints and strings) and not on Cool Objects.

Cool CPU Simulator

The normal Cool compiler executable (e.g., cool.exe) also serves as a Cool CPU Simulator that executes Cool Assembly Language programs. Just pass file.cl-asm as an argument.

The simulator performs the following actions:

1. Load the .cl-asm program into memory starting at address 1000. That is, if the first instruction in file.cl-asm is mov r1, r2, then memory location 1000 will hold the instruction mov r1, r2. If the second instruction in file.cl-asm is constant 55, then memory location 1001 will hold the integer 55.
2. Set sp and fp to 2,000,000,000. Remember, the stack starts at high addresses and grows down.
3. Search file.cl-asm for a label named start. The program counter is set to the address corresponding to that label. For example, if start : occurs just before the third instruction in file.cl-asm, then the program counter starts at 1002.
4. Fetch the instruction pointed to by the program counter and execute it. Unless the instruction specifies otherwise, the program counter is incremented by one and the process repeats.
5. When memory is allocated (e.g., by the alloc instruction), addresses starting from at least 20,000 are used.
6. If more than 1000 call instructions are executed before any return instructions are executed (i.e., if there are more than 1000 calls on the stack), the simulator terminates and prints a stack overflow error.

The constant values listed above (1000; 20,000; 2,000,000,000) should not be counted on by your program, but are listed here to help with debugging. Addresses near 1000 hold program instructions or compile-time data (i.e., the code segment), addresses near 20,000 hold the heap, and addresses near two billion are on the stack.

Debugging

Debugging assembly language programs is notoriously difficult! While writing your code generator, you will spend quite a bit of time running generated Cool Assembly programs through the Cool CPU Simulator to see if they work. Often they will not. The Cool CPU Simulator has been designed with a large number of features to aid debugging. Basically none of these features are present in traditional assemblers, so you actually have a wealth of debugging support, but it will still be difficult.

- The simulator tracks a notion of time -- the first instruction is executed at time one, the second at time two, etc. More importantly:
- The simulator tracks, for each register and memory value, the last time it was written to and the instruction that wrote to it. This can be invaluable for tracking down memory corruption errors (e.g., finding who is scribbling over memory) or otherwise determining why r1 holds an integer when you were sure it was supposed to hold a

string.

- If you try to read from a register or a memory address that has never been written to, the simulator will catch it and abort the program, rather than continuing with a garbage value.
- You can use the $\mathrm{debug\ r1}$ opcode to print out the current value of any register, as well as its last modification information.
- The simulator keeps track of integers, strings, labels and code segment addresses separately "under the hood". Thus if you execute $\mathrm{la\ r5\ my_label}$ and then inspect the value of $\mathrm{r5}$, it will print as $\mathrm{label\ my_label}$ rather than 1056 or whatever that address happens to be. This can be quite handy for tracking down problems related to virtual function tables. Perhaps more importantly:
- The simulator uses this type information when simulating instructions, and stops early if you provide the wrong type of argument. For example, in $\mathrm{st\ r1\ [\ 0\ ]\ <-\ r2}$, if $\mathrm{r1}$ is actually a string or a pointer to the code segment, the simulator will raise an error rather than silently corrupting your program. If $\mathrm{r1}$ is a label or integer address, everything works fine. (If the string example confuses you, remember that in Cool Assembly Language a raw string is a one-word value that fits in a register, not a C-style pointer to a buffer.)
- The simulator keeps a best effort stack trace. If you use the call and return instructions, the simulator will keep track of which functions were called, and from where, and print that back trace out if there is an error.
- When dynamically allocating memory, the simulator actually allocates more space than is needed and leaves the remainder empty. For example, if you make two allocations of five words each, you may get back the addresses 21,000 and 21,010. The range 21,005-21,009 remains unused, and if you attempt to read from it, the simulator will abort. This can help to prevent walking off the end of a buffer.
- If you attempt to divide by zero or dereference a null pointer, the simulator will catch it.
- Finally, if you use the $\mathrm{--trace-eval}$ option to $\mathrm{cool.exe}$ or execute the trace instruction (which toggles the state of tracing), the simulator will print copious debugging information before every time step, including the contents of all registers and the current instruction.

Control Flow Graphs

The Cool reference compiler also includes options to produce control flow graphic visualizations in the style of the dotty tool from the Graphviz toolkit.

Passing the $\mathrm{--cfg}$ option (with, for example, $\mathrm{--opt\ --asm}$) produces $\mathit{method}.\mathrm{dot}$, which can then be inspected via a number of tools. For example, this program:

```
class Main {
  main():Object {
    if (isvoid self) then
      (new IO).out_string("cannot happen!\n")
    else
      (new IO).out_string("hello, world!\n")
    fi
  };
};
```

Might produce this control-flow graph:

While you do not have to match the reference compiler exactly, inspecting its control-flow graphs can help you debug your own code to create control-flow graphs.

Performance Model

As discussed above, the Cool reference compiler also includes a reference machine simulator to interpret Cool Assembly Language instructions. This simulator can be invoked directly by passing a `.cl-asm` file to `cool.exe`:

```
cool$ cat hello-world.cl
class Main {
  main():Object {
    (new IO).out_string("hello, world!\n")
  };
};
cool$ ./cool --asm hello-world.cl
cool$ ./cool hello-world.cl-asm
hello, world!
```

The simulator can also give detailed performance information:

```
cool$ ./cool --profile hello-world.cl-asm
hello, world!
PROFILE:           instructions =        107 @    1 =>        107
PROFILE:        pushes and pops =         29 @    1 =>         29
PROFILE:             cache hits =         22 @    0 =>          0
PROFILE:           cache misses =        570 @  100 =>      57000
PROFILE:     branch predictions =          0 @    0 =>          0
PROFILE:  branch mispredictions =         11 @   20 =>        220
PROFILE:        multiplications =          0 @   10 =>          0
PROFILE:              divisions =          0 @   40 =>          0
PROFILE:           system calls =          2 @ 1000 =>       2000
CYCLES: 59356
```

The execution time of a Cool Assembly Language program is measured in simulated instruction cycles. In general, each assembly instruction takes one cycle. Some instructions, such as system calls or memory operation, can cost many more cycles. The total cycle cost of a program is the sum of all of its component cycle costs.

In modern architectures, memory hierarchy effects (e.g., caching) and branch prediction are dominant factors in the execution speed of a program. To give you a flavor for what real-world code optimization is like, the Cool Simulator also simulates a cache and a branch predictor.

The Cool Simulator features a 64-word least-recently-used fully associative combined instruction and data cache. It also uses a static backward = taken, forward = not taken branch prediction scheme.

We now discuss each of the performance components in turn:

1. **instructions**. Each Cool Assembly Language instruction executed costs at least one cycle. This represents the time taken to fetch and decode the instruction, as well as to shepherd it through the pipeline. Instructions such as `li`, `mov` and `add` take one cycle.
2. **pushes and pops**. Such `push` and `pop` involve both a load/store and also an add/sub, each costs an additional cycle (for a total of two). (`push` and `pop` can also incur cache miss penalties; see below.)
3. **cache hits & misses**. In modern computers, the CPU executes much faster than main memory: hundreds of "normal" instructions can be executed in the time it takes to fetch one value from memory. To mitigate this problem, a small number of values are placed in expensive, high-speed memory near the CPU. This small, fast memory stores recently-used values and is known as a **cache**. The Cool Simulator features a 64-word fully-associated cache: the values associated with 64 addresses can be accessed rapidly. If a memory read or write accesses an address that is in the cache, the instruction completes immediately with no extra cost. If a memory read or write accesses an address that is not in the cache, it costs 100 cycles while that value is read in from main memory. If there is no room in the cache to hold that new address's value, the address that has been touched (read or written) least recently is evicted and the new address/value is put in its place. Typical reasons for cache misses include compulsory, capacity and conflict. Note that the cache and cache miss penalty apply to every access to memory. This Includes:
    - Fetching the next instruction based on the program counter.
    - `push`, `pop`
    - `ld`, `st`
    - `IO.in_string`
    - `IO.out_string`
    - `String.length`
    - `String.concat` (three times)
    - `String.substr` (two times)
4. **branch prediction & misprediction**. In a modern pipelined CPU, the next instruction is fetched before the current instruction has completed. This means that the CPU needs to know the address of the next instruction as early as possible. For a conditional branch, that may be difficult: the CPU may have to wait until the comparison is complete to determine if the next instruction will be at `pc + 1` or `label`. Modern CPUs optimistically "guess" or "predict" that a branch will go one way or the other and then rollback instructions if they are wrong. A correctly-predicted branch costs nothing; a mispredicted branch costs 20 cycles. The following instructions are related to this cost:
    - `jmp` -- always correctly predicted
    - `call label` -- always correctly predicted
    - `bz bnz beq blt ble` -- The Cool CPU Simulator uses the following heuristic: if the address `label` is less than the address of the current PC (i.e., if label's definition occurs *before* the current PC in the assembly code), guess taken. Otherwise, guess not taken. This heuristic works well in practice: imagine a `for` loop that executes 10 times: the heuristic will be right 90% of the time.
    - `call reg` -- always mispredicted
    - `return` -- always mispredicted
5. **multiplication & division**. Integer multiplication and division take longer on most architectures than addition and subtraction. In the Cool Simulator, `mul` costs an extra

10 cycles and div costs an extra 40.

6. **system calls**. A system call involves trapping to the operating system, switching CPU protection contexts, putting the old process on the scheduling queue, handling the operation, rescheduling the new process, and switching CPU protection contexts again. System calls take forever. In the Cool Simulator, each $\mathrm{syscall}$ instruction takes 1000 extra cycles.

This cost model involves realistic components but potentially unrealistic values (e.g., a modern CPU would have a much larger non-associative cache, and also a much larger cache miss cost). If you're interested in that sort of performance modeling, take a graduate class in computer architecture. You should know that this CPU performance model is one of the most realistic that I've seen for a compiler optimization project in terms of the issues that it forces you to address.

The reference compiler includes a simple reference peephole optimizer, as well as a few optimizations backed by dataflow analyses (liveness, reaching definitions, constant folding) and register allocation enabled via the $--\mathrm{opt}$ flag. You can use it to get an idea for how to get started (but note that we are evil and strip all comments from the optimized output).

```
yuki:~/src/cool$ ./cool --opt --asm hello-world.cl
yuki:~/src/cool$ ./cool --profile hello-world.cl-asm
hello, world!
PROFILE:           instructions =         79 @    1 =>         79
PROFILE:        pushes and pops =         23 @    1 =>         23
PROFILE:             cache hits =         15 @    0 =>          0
PROFILE:           cache misses =        513 @  100 =>      51300
PROFILE:     branch predictions =          2 @    0 =>          0
PROFILE:  branch mispredictions =          7 @   20 =>        140
PROFILE:        multiplications =          0 @   10 =>          0
PROFILE:              divisions =          0 @   40 =>          0
PROFILE:           system calls =          2 @ 1000 =>       2000
CYCLES: 53542
```

For the $\mathrm{hello}-\mathrm{world}$ program, this optimizer reduces the cycle cost from 59356 to 53453 -- a 10% improvement. If you are writing an optimizer, you will want to do at least as well as the reference, averaged over many input programs. Notably, you'll probably want to implement much more than the required dead code elimination optimization.

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.

This version (used in Virginia Programming Language Design and Implementation courses) of Cool owes a great debt to George C. Necula and Bor-Yuh Evan Chang.

Footnotes

1. one
2. Nil. [[$Features$|2]] In this example, Nil is assumed to be a subtype of `List`.
3. inheritance. [[$Inheritance$|3]] Some object-oriented languages allow a class to inherit from multiple classes, which is equally aptly called "multiple inheritance."
4. object. ^4^ A shallow copy of a copies a itself, but does not recursively copy objects that a points to.
5. void. ^5^