# IC **Language Specification**

## The IC Language

For the implementation project, you will build a compiler for an object-oriented language called IC (for Irish Coffee[1]), which is essentially a simplified version of Java. Despite all of the simplifications allowing us to have a one-page grammar specification, this language preserves enough Java constructs and features to make this project challenging and to let us write many interesting programs. In the rest of this document we give a detailed description of the language, discussing its syntax and semantics.

IC and related assignments are based on similar assignments developed at Cornell University and Williams College.

## 1 Overview

The following list highlights the main features of the language:

- *Object oriented features*: it supports objects, single inheritance, subtyping, and virtual method calls; however, it does not support method overloading;
- *Types*: it is a strongly-typed language and provides: primitive types for integers, booleans, and strings; class types for objects; and array types.
- *Dynamic allocation and garbage collection*: it supports dynamic allocation for objects, strings and arrays. Such structures are always allocated on heap and variables hold references to them. The language also supports garbage collection for automatic de-allocation of heap space.
- *Run-time checks*: it supports run-time checks for null references, array bounds violations, and for negative array size allocations.

## 2 Lexical Considerations

Identifiers and keywords are case-sensitive. Identifiers must begin with an alphabetic character. Following the initial alphabetic character may be any sequence of alphabetic characters, numeric characters, or the underscore character (_). Uppercase and lowercase alphabetic characters are both considered alphabetic and are distinguished, so `x` and `X` are different identifiers. For simplicity, you may assume that class names always start with a capital letter and all other identifiers start with lower-case letters.

The following are the keywords in the language and cannot be used as identifiers:

```
class   extends  Library  void   int    boolean   string
return  if       else     while  break  continue  this
new     length   true     false  null
```

White spaces consists of a sequence of one or more space, tab, or newline characters. White spaces may appear between any tokens. Keywords and identifiers must be separated by white space or a token that is neither a keyword or an identifier. For instance, `elsex` represents a single identifier, not the keyword `else` followed by the identifier `x`.

Both forms of Java comments are supported. A comment beginning with the characters `//` indicates that the remainder of the line is a comment. A Java-style block comment is a sequence of characters that begins with `/*`, followed by any characters, including newline, up to the first occurrence of the end sequence `*/`. Unclosed comments are lexical errors.

[1] Or whatever else you can come with!

ML-style comments (with nesting) are also supported. They involve two lexical tokens: an open comment token `(*` and a close comment token `*)`. These comments support nesting. For example: `(* This is (* a single (* comment *) that *) ends (* after the last star-paren *) right here -> *)`

Integer literals may start with an optional negation sign `-`, followed a sequence of digits. Non-zero numbers should not have leading zeroes. Integers have 32-bit signed values between $-2^{31}$ and $2^{31} - 1$.

String literals are sequences of characters enclosed in double quotes. String characters can be:

1. printable ASCII characters (ASCII codes between decimal 32 and 126) other that quote `"` and backslash `\`, and
2. the escape sequences `\"` to denote quote, `\\` to denote backslash, `\t` to denote tab, and `\n` for newline.

No other characters or character sequences can occur in a string. Unclosed strings are lexical errors.

Boolean literals must be one of the keywords `true` or `false`. The only literal for heap references is `null`.

# 3 Program Structure

A program consists of a sequence of class declarations. Each class in the program contains field and method declarations. A program must have exactly one method `main`, with the following signature:

```
void main (string[] args) { ... }
```

In contrast to Java, the main method is dynamic. That is, if `main` belongs to a class `A`, the execution of the program consists of creating a new object of class `A`, then invoking its `main` method. In fact, all of the method calls (except library function calls) are dynamic, which simplifies typechecking in this language.

# 4 Variables

Program variables may be local variables or parameters of methods, in which case they are allocated on stack; or they may be fields of objects or array elements, allocated on heap. Program variables of type int and boolean hold integer and boolean values, respectively. Variables of other types contain references to heap-allocated structures (i.e., strings, arrays, or objects).

The program does not initialize variables by default when they are declared. Instead, the compiler statically checks that each variable is guaranteed to have a value assigned before being used. Object fields and array elements are initialized with default values (0 for integers, `false` for booleans, and `null` for references) when such structures are dynamically created.

The language allows variables to be initialized when declared at the beginning of a statement block. The initialization expression can refer to variables in enclosing scopes (or to the formal arguments of the enclosing method), but cannot involve any of the variables declared in the current block.

# 5 Strings

For string references, the language uses the primitive type string (unlike Java, where String is a class). Strings are allocated on heap and are immutable, which means that the program cannot modify their contents. The language allows only the following operations on string variables: assignments of string references (including `null`); concatenating strings with the `+` operator; and testing for string reference equality using `==` and `!=` (Note: this operator does not compare string contents). Library functions are provided to convert integers to strings, etc.

# 6 Arrays

The language supports arrays with arbitrary element types. If `T` is a type , then `T[]` is the type for an array with elements of type `T`. In particular, array elements can be arrays themselves, allowing programmers to

build multidimensional arrays. For instance, the type `T[][]` describes a two-dimensional array constructed as an array of array references `T[]`.

Arrays are dynamically created using the `new` construct: `new T[n]` allocates an array of type `T` with `n` elements and initializes the elements with their default values. The expression `new T[n]` yields reference to the newly created array. Arrays of size $n$ are indexed from 0 to $n-1$ and the standard bracket notation is used to access array elements. If the expression `a` is a reference to an array of length $n$, then `a.length` is $n$, and `a[i]` returns the `(i+1)`-th element in the array. For each array access `a[i]`, the program checks at run-time that `a` is not null and that the access is within bounds: $0 \leq \texttt{i} < \texttt{n}$. Violations will terminate the program with an error message.

# 7 Classes

Classes are collections of fields and methods. They are defined using declarations of the form:

`class A extends B {` *body* `}`

where *body* is a sequence of field and method declarations. The `extends` clause is optional. When the extends clause is present, class `B` inherits all of the features (methods and fields) of class `A`. Only one class can be inherited. Hence IC supports only single inheritance. We say that `A` is a subclass of `B`, and that `B` is a superclass of `A`.

Classes can only extend previously defined classes. In other words, a class cannot extend itself or another class defined later in the program. This ensures that the class hierarchy has a tree structure.

Method overloading is not supported: a class cannot have multiple methods with the same name, even if the methods have different number of types of arguments, or different return types. Hidden fields are not permitted either. All of the newly defined fields in a subclass must have different names than those in the superclasses.

However, methods can be overridden in subclasses. Subclasses can re-define more specialized versions of methods defined in their superclasses.

The language does not permit hidden fields either. That is, for each class declaration `class B extends A`, all of the newly defined fields of `B` must have different names than the inherited fields from `A`. Otherwise, those inherited fields couldnt be accessed by the methods of B.

# 8 Subtyping

Inheritance induces a subtyping relation. When class `A` extends class `B`, `A` is a subtype of `B`, written $\texttt{A} \leq \texttt{B}$. Subtyping is also reflexive and transitive. Additionally, the special type Null is a subtype of any reference type:

$$\frac{\texttt{A extends B } \{\ldots\}}{\texttt{A} \leq \texttt{B}} \qquad \frac{}{\texttt{A} \leq \texttt{A}} \qquad \frac{\texttt{A} \leq \texttt{B} \quad \texttt{B} \leq \texttt{C}}{\texttt{A} \leq \texttt{C}} \qquad \frac{}{\mathsf{Null} \leq \texttt{A}} \qquad \frac{}{\mathsf{Null} \leq \texttt{T[]}} \qquad \frac{}{\mathsf{Null} \leq \texttt{string}}$$

If `A` is a subtype of `B`, a value of type `A` can be used in the program whenever the program expects a value of type `B`.

Subtyping is not covariant for array types: if `A` is a subtype of `B` then `A[]` is *not* a subtype of `B[]`. Instead, array subtyping is type invariant, which means that each array type is only a subtype of itself.

As in Java, subtyping for virtual methods is type invariant, both for the parameters and for the return value. This means that each virtual method in a subclass must have the same number and type of arguments, and the same return type as the corresponding methods from the superclasses.

# 9 Objects

Objects of a certain class can be dynamically created using the `new` construct. If `A` is a declared class, `new A()` allocates an object of class `A` on heap and initializes all of its fields with the default values. The expression `new A()` yields a reference to the newly allocated object.

Object fields and instance methods are accessed using the '`.`' symbol. The expression `o.f` denotes the field `f` of object `o`, and the expression `o.m()` denotes a call to the virtual method `m` of object `o`. The keyword `this` refers to the current object (i.e., the object on which the current method is invoked).

For virtual methods, the actual method being invoked by `o.m()` cannot be determined statically, because the precise type of receiver object `o` is unknown – it can be the declared class for `o` or any of its subclasses. Virtual method calls are resolved at run-time via dynamic dispatch.

Object references have class types: each definition `class A` introduces a class type `A`. Class types can then be used in declarations for reference variables. For instance, "`A obj`" declares a variable `obj` of type `A`, that is, a reference to an object of class `A`.

A class name `A` can be used as the type of an object reference anywhere in the program. In particular, it can appear in the body of the `class A` declaration itself, or even before that, as in the following example. This allows to build recursive and mutually recursive class structures, such as the ones below:

```
class List { int data; List next; }
class Node { int data; Edge[] edges; }
class Edge { itn label; Node dest; }
```

# 10 Method Invocation

A method invocation consists of the following steps: passing the parameter values from the caller to the callee, executing the body of the callee, and returning the control and the result value (if any) to the caller.

At each method invocation site, the program evaluates the expressions representing the actual arguments and then assigns the computed values to the corresponding formal parameters of the method. Object, array, or string arguments are passed as references to such structures. The arguments are always evaluated from left to right.

After parameters are assigned values, the program executed the body of the invoked method. When the execution reaches a return statement or reaches the end of the method body, the program transfers the control back to the caller. If the return statement has an expression argument, that argument is evaluated and the computed value is also returned to the caller.

At each method invocation, the number and types of actual values of the call site must be the same as the number and types of formal parameters in the method declaration.

Also, the return type from the declaration of a method must match the return statements in the body of that method. More precisely, if a method is declared to return void, then return statements in the method body should have no return expression. In this case, the method is allowed to reach the end its body without encountering a return statement. Otherwise, if the method is declared with a return type `T`, then return statements must have a return value of type `T`. In this case, the method body is required to execute a return statement before it reaches the end of its body.

# 11 Scoping Rules

For each program, there is a hierarchy of scopes consisting of: the global scope, the class scopes, the method scopes, and the local scopes for blocks within each method. The global scope consists of the names of all classes defined in the program. The scope of a class is the set of fields and methods of that class. The scopes of subclasses are nested inside the scopes of their superclasses. The scope of a method consists of the formal parameters and local variables defined in the block representing the body of the method. Finally, a block scope contains all of the variables defined at the beginning of that block. When resolving an identifier at a certain point in the program, the enclosing scopes are searched for that identifier.

There are a couple of scope rules. First, identifiers can only be used if they are defined in one of the enclosing scopes. More precisely, variables can only be used (read or written) after they are defined in one of the enclosing block or method scopes. Fields and methods can be used in expressions of the form `expr.f` or `expr.m` when the object designator `expr` has class type `T` and the scope of `T` contains those fields and methods. This means that all methods and fields are public and can be accessed by other classes. Finally, class names can be used anywhere, provided they are defined in the program (either before or after the point

where they are referred to). Fields and methods that are defined in the enclosing class can also be acceessed simply as `f` or `m`, in which case the object designator `this` is implicit.

Another rule is that identifiers (classes, fields, methods, and variables) cannot be defined multiple times in the same scope, unless they are of different kinds (that is, a field and a method of the same class can have the same name). Also, fields with the same names cannot be re-defined in nested class scopes (i.e., in subclasses). Otherwise, identifiers can be defined multiple times in different, possibly nested, scopes. For variables, inner scopes shadow outer scopes. That is, if variables with the same name occur in nested scopes, then each occurence of that variable name refers to the variable in the innermost scope. Finally, it is not allowed to shadow method parameters — the local variables must have different names than the parameters of the enclosing method.

The following examples illustrate some aspects of these scoping rules:

- This program is legal, since '`f`' can act as both a field and a method name in the same class:

```
class A {
  int f;
  void f() { }
}
```

- The following two programs are also legal:

```
class A {
  int x;
  void f() {
    int x;
    x = 1; // here x refers to the local variable x
    this.x = 1; // here x refers to the field x
  }
}
```

```
class A {
  void f() {
    int x;
    {
      boolean x;
      x = true; // x refers to the variable defined in the inner scope.
    }
  }
}
```

- Shadowing method parameters with local variables is illegal, as in the following:

```
class A {
  void f(int x) {
    int x = 1;  // illegal
  }
}
```

- The following code is also illegal, since fields cannot be redefined in subclasses:

```
class A {
  int x;
}

class B extends A {
  int x;
}
```

## 12 Statements

IC has the standard control statements: assignments, method calls, `return` statements, `if` constructs, `while` loops, `break` and `continue` statements, and statement blocks.

Each assignment statement $l$ = e updates the location represented by $l$ with the value of expression $e$. The updated location $l$ can be a local variable, a parameter, a field, or an array element. The type of the updated location must match the type of the evaluated expression. For integers and booleans, the assignment copies the integer or boolean value. For string, array, or object types, the assignment only copies the reference.

Method invocations can be used as statements, regardless of whether they return values or not. If the invoked method returns a value, that value is discarded.

The `if` statement has the standard semantics. It first evaluates the test expression, and executes one of its branches depending on whether the test is true of false. The else clause of an if statement always refers to the innermost enclosing if.

The `while` statement executes its body iteratively. At each iteration, it evaluates the test condition. If the condition is false, then it finishes the execution of the loop; otherwise it executes the loop body and continues with the next iteration. The `break` and `continue` statements must occur in the body of an enclosing loop in the current method. The `break` statement terminates the loop and the execution continues with the next statement after the loop body. The `continue` statement terminates the current loop iteration; the execution of the program proceeds to the next iteration and tests the loop condition. When `break` and `continue` statements occur in nested loops, they refer to the innermost loop.

Blocks of statements consist of a sequences of statements and variable declarations. Blocks are statements themselves, so blocks and statements can be nested arbitrarily deep.

## 13 Expressions

Program expressions include:

- memory locations: local variables, parameters, fields, or array elements;
- calls to methods with non-void return types;
- the current object `this`;
- new object or array instances, created with `new T()` or `new T [` $e$ `]`;
- the array length expression $e$`.length`;
- unary or binary expressions; and
- integer, string, and `null` literals.
- any expression enclosed in parentheses, to make operator precedence explicit.

## 14 Operators

Unary and binary operators include the following:

- Arithmetic operators: addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulo `%`. The operands must be integers. Division by zero and modulus of zero are dynamically checked, and cause program termination.
- Relational comparison operators: less than `<`, less or equal than `<=`, greater than `>`, and greater or equal then `>=`. Their operands must be integers.
- Equality comparison operators: equal `==` or different `!=`. The operands must have the same type. For integer and boolean types, operand values are compared. For the other types, references are compared.

- Conditional operators: short-circuit "and", `&&`, and short-circuit "or", `||`. If the first operand of `&&` evaluates to false, its second operand is not evaluated. Similarly, if the first operand of `||` evaluates to true, its second operand is not evaluated. The operands must be booleans.
- unary operators: sign change `-` for integers and logical negation `!` for booleans.

The operator precedence and associativity is defined by the table below. Here, priority 1 is the highest, and priority 9 is the lowest.

| Priority | Operator | Description | Associativity |
|---|---|---|---|
| 1 | `[]` `()`<br>`.` | array index, method call<br>field/method access | left |
| 2 | `-` `!` | unary minus, logical negation | right |
| 3 | `*` `/` `%` | multiplication, division, remainder | left |
| 4 | `+` `-` | addition, subtraction | left |
| 5 | `<` `<=` `>` `>=` | relational operators | left |
| 6 | `==` `!=` | equality comparison | left |
| 7 | `&&` | short-circuit and | left |
| 8 | `\|\|` | short-circuit or | left |
| 9 | `=` | assignment | right |

# 15 IC Syntax

The language syntax is show in Figure 1. Here, keywords are shown using typewriter fonts (e.g., `while`); operators and punctuation symbols are shown using single quotes (e.g., '`;`'); the other terminals are written using small caps fonts (Id, ClassId, Integer, and String); and nonterminals using slanted fonts (e.g., *formals*). The remaining symbols are meta-characters: (...)* denotes the Kleene star operation and [...] denotes an optional sequence of symbols.

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{classDecl}^* \\
\textit{classDecl} & ::= & \texttt{class}\ \textsc{ClassId}\ [\texttt{extends}\ \textsc{ClassId}]\ \texttt{'\{'}\ (\textit{fieldDecl} \mid \textit{methodDecl})^*\ \texttt{'\}'} \\
\textit{fieldDecl} & ::= & \textit{type}\ \textsc{Id}\ (\texttt{','}\ \textsc{Id})^*\ \texttt{';'} \\
\textit{methodDecl} & ::= & (\textit{type} \mid \texttt{void})\ \textsc{Id}\ \texttt{'('}\ [\textit{formals}]\ \texttt{')'}\ \textit{block} \\
\textit{formals} & ::= & \textit{type}\ \textsc{Id}\ (\texttt{','}\ \textit{type}\ \textsc{Id})^* \\
 & & \\
\textit{type} & ::= & \texttt{int} \mid \texttt{boolean} \mid \texttt{string} \mid \textsc{ClassId} \mid \textit{type}\ \texttt{'['}\ \texttt{']'} \\
 & & \\
\textit{block} & ::= & \texttt{'\{'}\ \textit{varDecl}^*\ \ \textit{stmt}^*\ \texttt{'\}'} \\
\textit{varDecl} & ::= & \textit{type}\ \ \textsc{Id}\ [\texttt{'='}\ \textit{expr}]\ \ (\texttt{','}\ \textsc{Id}\ [\texttt{'='}\ \textit{expr}])^*\ \ \texttt{';'} \\
 & & \\
\textit{stmt} & ::= & \textit{location}\ \texttt{'='}\ \textit{expr}\ \ \texttt{';'} \\
 & \mid & \textit{call}\ \ \texttt{';'} \\
 & \mid & \texttt{return}\ [\textit{expr}]\ \ \texttt{';'} \\
 & \mid & \texttt{if}\ \texttt{'('}\ \textit{expr}\ \texttt{')'}\ \textit{stmt}\ \ [\texttt{else}\ \textit{stmt}] \\
 & \mid & \texttt{while}\ \texttt{'('}\ \textit{expr}\ \texttt{')'}\ \textit{stmt} \\
 & \mid & \texttt{break}\ \texttt{';'} \\
 & \mid & \texttt{continue}\ \texttt{';'} \\
 & \mid & \textit{block} \\
 & & \\
\textit{expr} & ::= & \textit{location} \\
 & \mid & \textit{call} \\
 & \mid & \texttt{this} \\
 & \mid & \texttt{new}\ \textsc{ClassId}\ \texttt{'('}\ \texttt{')'} \\
 & \mid & \texttt{new}\ \textit{type}\ \texttt{'['}\ \textit{expr}\ \texttt{']'} \\
 & \mid & \textit{expr}\ \texttt{'.'}\ \texttt{length} \\
 & \mid & \textit{expr binop expr} \\
 & \mid & \textit{unop expr} \\
 & \mid & \textit{literal} \\
 & \mid & \texttt{'('}\ \textit{expr}\ \texttt{')'} \\
 & & \\
\textit{call} & ::= & \textit{libCall} \mid \textit{virtualCall} \\
\textit{libCall} & ::= & \texttt{Library}\ \texttt{'.'}\ \textsc{Id}\ \texttt{'('}\ [\textit{expr}\ \ (\texttt{','}\ \textit{expr})^*]\ \ \texttt{')'} \\
\textit{virtualCall} & ::= & [\textit{expr}\ \texttt{'.'}]\ \textsc{Id}\ \texttt{'('}\ [\textit{expr}\ \ (\texttt{','}\ \textit{expr})^*]\ \ \texttt{')'} \\
\textit{location} & ::= & \textsc{Id} \mid \textit{expr}\ \texttt{'.'}\ \textsc{Id} \mid \textit{expr}\ \texttt{'['}\ \textit{expr}\ \texttt{']'} \\
 & & \\
\textit{binop} & ::= & \texttt{'+'} \mid \texttt{'-'} \mid \texttt{'*'} \mid \texttt{'/'} \mid \texttt{'\%'} \mid \texttt{'\&\&'} \mid \texttt{'||'} \\
 & \mid & \texttt{'<'} \mid \texttt{'<='} \mid \texttt{'>'} \mid \texttt{'>='} \mid \texttt{'=='} \mid \texttt{'!='} \\
\textit{unop} & ::= & \texttt{'-'} \mid \texttt{'!'} \\
\textit{literal} & ::= & \textsc{Integer} \mid \textsc{String} \mid \texttt{true} \mid \texttt{false} \mid \texttt{null}
\end{array}
$$

Figure 1: IC Syntax

# 16 Typing Rules

**Typing rules for expressions.**

$$\frac{}{E \vdash \texttt{true} : \mathsf{boolean}} \qquad \frac{}{E \vdash \texttt{false} : \mathsf{boolean}}$$

$$\frac{}{E \vdash \textit{integer-literal} : \mathsf{int}} \qquad \frac{}{E \vdash \textit{string-literal} : \mathsf{string}}$$

$$\frac{E \vdash e_0 : \mathsf{int} \quad E \vdash e_1 : \mathsf{int} \quad op \in \{\texttt{+},\texttt{-},\texttt{/},\texttt{*},\texttt{\%}\}}{E \vdash e_0\ op\ e_1 : \mathsf{int}} \qquad \frac{E \vdash e_0 : \mathsf{string} \quad E \vdash e_1 : \mathsf{string}}{E \vdash e_0\ +\ e_1 : \mathsf{string}}$$

$$\frac{E \vdash e_0 : T_0 \quad E \vdash e_1 : T_1 \quad T_0 \le T_1 \text{ or } T_1 \le T_0 \quad op \in \{\texttt{==},\texttt{!=}\}}{E \vdash e_0\ op\ e_1 : \mathsf{boolean}} \qquad \frac{E \vdash e_0 : \mathsf{int} \quad E \vdash e_1 : \mathsf{int} \quad op \in \{\texttt{<=},\texttt{<},\texttt{>=},\texttt{>}\}}{E \vdash e_0\ op\ e_1 : \mathsf{boolean}}$$

$$\frac{E \vdash e_0 : \mathsf{boolean} \quad E \vdash e_1 : \mathsf{boolean} \quad op \in \{\texttt{\&\&},\texttt{||}\}}{E \vdash e_0\ op\ e_1 : \mathsf{boolean}} \qquad \frac{E \vdash e : \mathsf{int}}{E \vdash \texttt{-}e : \mathsf{int}}$$

$$\frac{E \vdash e_0 : T\texttt{[]} \quad E \vdash e_1 : \mathsf{int}}{E \vdash e_0[e_1] : T} \qquad \frac{E \vdash e : \mathsf{boolean}}{E \vdash \texttt{!}e : \mathsf{boolean}}$$

$$\frac{E \vdash e : T\texttt{[]}}{E \vdash e\,\texttt{.length} : \mathsf{int}} \qquad \frac{E \vdash e : \mathsf{int}}{E \vdash \texttt{new}\ T[e] : T\texttt{[]}}$$

$$\frac{}{E \vdash \texttt{new}\ T() : T} \qquad \frac{}{E \vdash \texttt{null} : \mathsf{Null}}$$

$$\frac{id : T \in E}{E \vdash id : T} \qquad \frac{E \vdash e : C \quad (id : T) \in C}{E \vdash e\,.\,id : T}$$

$$\frac{E \vdash e_0 : T_1 \times \ldots \times T_n \to T_r \quad E \vdash e_i : T_i' \quad T_i' \le T_i \quad \text{for all } i = 1..n}{E \vdash e_0(e_1, \ldots, e_n) : T_r} \qquad \frac{\texttt{Library}.m \textit{ has type } T_1 \times \ldots \times T_n \to T_r \quad E \vdash e_i : T_i' \quad T_i' \le T_i \quad \text{for all } i = 1..n}{E \vdash \texttt{Library}.m(e_1, \ldots, e_n) : T_r}$$

**Typing rules for statements.**

$$\frac{E \vdash e_l : T \quad E \vdash e : T' \quad T' \le T}{E \vdash e_l = e} \qquad \frac{E \vdash e : \mathsf{boolean} \quad E \vdash S_1 \quad E \vdash S_2}{E \vdash \texttt{if}\ (e)\ \texttt{then}\ S_1\ \texttt{else}\ S_2}$$

$$\frac{E \vdash e : \mathsf{boolean} \quad E \vdash S}{E \vdash \texttt{while}\ (e)\ S} \qquad \frac{}{E \vdash \texttt{break}\ ;} \qquad \frac{}{E \vdash \texttt{continue}\ ;}$$

$$\frac{E \vdash e_0(e_1, ..., e_n) : T}{E \vdash e_0(e_1, ..., e_n)\ ;} \qquad \frac{E \vdash e : T \quad \texttt{ret} : T' \in E \quad T \le T'}{E \vdash \texttt{return}\ e\ ;} \qquad \frac{\texttt{ret} : \mathsf{void} \in E}{E \vdash \texttt{return}\ ;}$$

$$\frac{E \vdash e : T' \quad T' \le T \quad E, x : T \vdash S}{E \vdash T\ x = e;\ S} \qquad \frac{E, x : T \vdash S}{E \vdash T\ x;\ S} \qquad \frac{S_1 \text{ not declaration} \quad E \vdash S_1 \quad E \vdash S_2}{E \vdash S_1\ \ S_2}$$

**Type rules for class and method declarations.**

$$\frac{\begin{array}{c} \mathit{classes}(P) = C_1, ..., C_n \\ \vdash C_i \ \text{ for all } i = 1..n \end{array}}{\vdash P} \ [\textsc{Program}]$$

$$\frac{\begin{array}{c} \mathit{methods}(C) = m_1, ..., m_k \\ \mathit{Env}(C, m_i) \vdash m_i \ \text{ for all } i = 1..k \end{array}}{\vdash C} \ [\textsc{Class}]$$

$$\frac{E, x_1 : t_1, ..., x_n : t_n, \texttt{ret} : t_r \vdash S_{body}}{E \vdash t_r \ m(t_1 \ x_1, ..., t_n \ x_n) \ \{ \ S_{body} \ \}} \ [\textsc{Method}]$$

Here, *Env(C,m)* is the environment (or scope) for class $C$ and method $m$. Thus *Env(C,m)* contains all of the methods and fields of $C$, including those declared in $C$'s superclasses. Finally, *classes(C)* yields all of the classes in program $P$; and *methods(C)* yields the declarations of all methods in the body of class $C$ (but not those inherited from other classes).

# 17 Library Functions

IC uses a simple mechanism to support I/O operations, datatype conversions, and other system-level functionality. The signatures of all of these functions are declared as follows:

| `Library` Method | Behavior |
|---|---|
| `void println(string s);` | prints string s followed by a newline |
| `void print(string s);` | prints string s |
| `void printi(int i);` | prints integer i |
| `void printb(boolean b);` | prints boolean b |
| | |
| `int readi();` | reads one character from the input |
| `string readln();` | reads one line from the input |
| `boolean eof();` | checks end-of-file on standard input |
| | |
| `int stoi(string s, int n);` | returns the integer that s represents or n of s is not an integer |
| `string itos(int i);` | returns a string representation of i |
| `int[] stoa(string s);` | an array with the ascii codes of chars in s |
| `string atos(int[] a);` | builds a string from the ascii codes in a |
| | |
| `int random(int n);` | returns a random number between 0 and n-1 |
| `int time();` | number of milliseconds since program start |
| `void exit(int n);` | terminates the program with exit code n |

To invoke library functions, the program must use method calls qualified with the `Library` name; for instance, `Library.random(100)` or `Library.stoi(412,0)`. For simplicity, there will be no interfaces for library functions and library calls do not need to be type-checked. For each qualified call `Library.f`, the compiler will generate a static call to `f` in the assembly output, regardless of whether `f` is available in the library or not. Users can extend the standard library `libic.a` with additional functions and invoke them using qualified calls in the source program, but one must take care to invoke them with the right number and type of arguments.