IC Language Specification

For the CS413 project, you will build a compiler for an object-oriented language called IC (for Irish Coffee), which is essentially a simplified version of Java. Despite all the simplifications which allowed us to have a one-page grammar specification, the language preserves enough Java constructs and features to make this project challenging and to let us write many interesting programs. Figure 1 presents the full grammar of the language. In the rest of this document we give a detailed description of the language, discussing its syntax and semantics.

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.

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

class	extends	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 will represent 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. In addition, a comment can be a sequence of characters that begins with /*, followed by any characters, including newline, up to the first occurrence of the end sequence */. Unclosed comments, which start with the sequence /*, but don't have a closing sequence */, are considered lexical errors.

Integer literals may start with an optional negation sign -, followed by either 0, or a digit in the range 1-9 and a sequence of zero or more digits in the range 0-9. Integers are 32-bit values between -2^{31} and $2^{31} - 1$ (that is, decimal values between -2147483648 and 2147483647).

String literals are sequences of characters enclosed in double quotes. The sequence of characters in a string consists of: 1) printable ASCII characters (ASCII codes between decimal 32 and 126) other and 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 may occur in a string. Unclosed strings, which start with a quote ", but don't have a closing unescaped quote ", are considered lexical errors.

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

Program structure

A program consists of a sequence of class declarations. Each class in the program contains of a sequence of 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 running its main method. In fact, all of the method calls (except library function calls) are dynamic, which simplifies typechecking in this language.

Program 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.

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) to string variables; concatenation of strings, using the + sign; and comparison operations, to compare string references (not the string contents).

Arrays

The language allows programmers to define arrays with arbitrary element types: if T is a type , then T[] is the type of arrays with elements of type T. Note that the syntax for array types is recursive, so that the array elements can be arrays themselves. Programmers can use such structures to build multidimensional arrays. For instance, the type T[][] describes a two-dimensional array constructed as an array of array references of type 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] evaluates to a 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. The language also provides a construct to obtain the length of an array. 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 language provides run-time checks to ensure that a is not null and that the access is within bounds: $0 \leq i < n$. If the array access is out of bounds, the program terminates with an error message.

Classes

This language provides two standard features of object-oriented languages: classes and inheritance. Classes are collections of fields and methods. They are defined using declarations of the form: class A { *body* }, where *body* is a sequence of field and method declarations.

Classes may optionally extend other existing classes, using the standard inheritance mechanism in object-oriented languages. Our language allows each class to extend at most one other class, that is, it only supports single inheritance. Classes are inherited using declarations of the form: class A extends B {*body* }. Here, class A inherits all of the fields and methods of B; it also adds the new fields and methods specified in the declaration of A. We say that A is a subclass (or derived class) of B, and that B is a superclass (or base class) of A. In general, we define the subclass relation as the transitive closure of the direct subclass relation described above. That is, if A is a subclass of B, and B is a subclass of C, then A is a subclass of C.

We require that classes extend only 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.

The language doesn't support method overloading. 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. We require all methods in each class to have different names.

The language doesn't 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 B. Otherwise, those inherited fields couldn't be accessed by the methods of B.

However, subclasses can re-define methods already defined by their superclasses. In other words, subclasses can define more specialized versions of existing methods, which access the additional fields and methods of the subclass. The specialized methods are called virtual methods.

Subtyping

Class inheritance via the extends construct defines a subtype relation between their types. If class B extends A, then B is a subtype of A. Subtyping is also reflexive (each class type is a subtype of itself) and is antisymmetric (there are no distinct types which are subtypes of each other). Hence, the subtyping relation is a partial order.

Subtyping guarantees the following property: if B is a subtype of A, then B has all of the features (i.e., fields and methods) of A. Therefore, a value of type B can be used in the program whenever a value of type A is expected to be read, for instance in the function arguments or in the right hand side of an assignment. In contrast to Java, subtyping is not covariant for array types: if A is a subtype of B then A[] is *not* a subtype of B[]. In our language, subtyping on arrays is invariant, which means that each array type is a subtype of itself only.

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.

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 methods are accessed using '.': the expression o.f denotes the field f of object o, and the expression o.m() denotes a call to a 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. To resolve virtual method calls, the language uses a run-time dynamic dispatch mechanism which dynamically checks the run-time time of object o and invokes the appropriate method.

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, provided that the program contains (possibly later) a declaration of class A. The language doesn't require the class type A to be used before the declaration of class A; in particular, the type A can appear in the body of the class A declaration itself, or even before that (as a forward reference). This allows programmers to build recursive and mutually recursive class structures, such as:

```
class List { int d; List next; }
class Node { int d; Edge[] edges; }
class Edge { Node to, from; }
```

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.

Scope 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 either directly (without any prefix) if the scope of the current class contains those fields or method, or they can be used in expressions of the form *expr.f* or *expr.m* when *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).

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.

Statements

The statements in this language are: assignments, method calls, return statements, if constructs, while loops, break and continue statements, or blocks of statements.

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 method parameter, an object field, or an array element. The type of the updated location must match the type of the evaluated expression. For integers and boolean types, the assignment copies the integer or boolean value. For string, array, or object types, the assignment only copies the reference to the object (i.e., the object itself is not copied).

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

The if statement has the standard semantics. It consists of a conditional test expression of type boolean, a true branch, and an optional false branch. It first evaluates the test expression. If this expression evaluates to true, it executes the true branch of the statement. If the expression evaluates to false and the else clause is present, that branch is executed; otherwise, the control is transferred to the next statement after the if. The else clause always refers to the innermost enclosing if statement.

The while statement also has the standard semantics. It consists of a conditional test expression of type boolean, and a loop body. The loop construct executes iteratively. At each iteration, it evaluates the test condition. If the condition evaluates to 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 sequence of local variable declarations, followed by a sequence of statements. Since blocks are statements themselves, it means that blocks and statements can be nested.

Expressions

An expression may be one of the following:

- The value of a memory location: local variable, function parameter, object field, or array element.

- Value returned by method call. The invoked method must have a declared type `T` different than `void`.
- The current object `this`. The type of `this` is the enclosing class.
- A `new` construct, which creates an object or an array. In the expression `new T()`, type `T` must be an object type. In the expression `new T[`*expr*`]`, *expr* must have integer type and `T` can be any type.
- The expression *expr*.`length` representing the length of an array. Here, *expr* must have array type.
- A unary or binary expression constructed with the operators presented below.
- A literal, as described in the section about lexical considerations.
- Expression enclosed in parentheses, to change or make the operator precedence explicit.

Operators

Unary and binary operators include:

- Arithmetic operators: addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulo `%`. All of these operate only on integers and yield integer values, with the exception of `+` which is also used to concatenate two strings. A divisor or modulus of zero is checked, and results in the program terminating.
- Relational comparison operators: less than `<`, less or equal than `<=`, greater than `>`, and greater or equal then `>=`. These operands for relational operators must be integer expressions.
- Equality comparison operators: equal `==` or different `!=`. The operands must have the same type. For integer and boolean types, operand values are compared. For other types, references are compared.
- Conditional operators include the short-circuit and operator `&&`, and the short-circuit or operator `||`. These have the same semantics as in Java. 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 of the short-circuit operators must be booleans.
- unary operators: sign change `-` for integers and logical negation `!` for booleans.

The following table indicates the operator precedence and associativity:

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

Typing rules for expressions

$$\frac{}{A \vdash \textsf{true} : \textsf{boolean}} \qquad \frac{}{A \vdash \textsf{false} : \textsf{boolean}}$$

$$\frac{}{A \vdash \textit{integer-literal} : \textsf{int}} \qquad \frac{}{A \vdash \textit{string-literal} : \textsf{string}}$$

$$\frac{A \vdash E_0 : \textsf{int} \quad A \vdash E_1 : \textsf{int}}{A \vdash E_0 \ op \ E_1 : \textsf{int}} \ op \in \{\texttt{+}, \texttt{-}, \texttt{/}, \texttt{*}, \texttt{\%}\} \qquad \frac{A \vdash E_0 : \textsf{string} \quad A \vdash E_1 : \textsf{string}}{A \vdash E_0 \ + \ E_1 : \textsf{string}}$$

$$\frac{A \vdash E_0 : T \quad A \vdash E_1 : T}{A \vdash E_0 \ op \ E_1 : \textsf{boolean}} \ op \in \{\texttt{==}, \texttt{!=}\} \qquad \frac{A \vdash E_0 : \textsf{int} \quad A \vdash E_1 : \textsf{int}}{A \vdash E_0 \ op \ E_1 : \textsf{boolean}} \ op \in \{\texttt{<=}, \texttt{<}, \texttt{>=}, \texttt{>}\}$$

$$\frac{A \vdash E_0 : \textsf{boolean} \quad A \vdash E_1 : \textsf{boolean}}{A \vdash E_0 \ op \ E_1 : \textsf{boolean}} \ op \in \{\texttt{\&\&}, \texttt{||}\} \qquad \frac{A \vdash E : \textsf{int}}{A \vdash \texttt{-}E : \textsf{int}}$$

$$\frac{A \vdash E_0 : T\texttt{[]} \quad A \vdash E_1 : \textsf{int}}{A \vdash E_0[E_1] : T} \qquad \frac{A \vdash E : \textsf{boolean}}{A \vdash \texttt{!}E : \textsf{boolean}}$$

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

$$\frac{T \in C}{A \vdash \textsf{new}\ T\texttt{()} : T} \qquad \frac{\begin{array}{c} A \vdash E_0 : T_1 \times \ldots \times T_n \to T \\ A \vdash E_i : T_i,\ 1 \le i \le n \end{array}}{A \vdash E_0(E_1, \ldots, E_n) : T}$$

$$\frac{id : T \in A}{A \vdash id : T} \qquad \frac{A \vdash E : T \quad T \in C \quad (id : T') \in T}{A \vdash E\,.\,id : T'}$$

Typing rules for statements

$$\frac{A \vdash E : T' \quad T' <: T \quad A, \mathsf{x} : T \vdash S : \textsf{unit}}{A \vdash T\ \mathsf{x} = E\ ;\ S : \textsf{unit}} \ (\textsc{Decl1}) \qquad \frac{A, \mathsf{x} : T \vdash S : \textsf{unit}}{A \vdash T\ \mathsf{x}\ ;\ S : \textsf{unit}} \ (\textsc{Decl2})$$

$$\frac{A \vdash S_1 : \textsf{unit} \quad S_1 \text{ not declaration} \quad A \vdash S_2 : \textsf{unit}}{A \vdash S_1\ ;\ S_2 : \textsf{unit}} \ (\textsc{Seq}) \qquad \frac{A \vdash L : T \quad A \vdash E : T' \quad T' <: T}{A \vdash L = E : \textsf{unit}} \ (\textsc{Assign})$$

$$\frac{A \vdash E : \textsf{boolean} \quad A \vdash S_1 : \textsf{unit} \quad A \vdash S_2 : \textsf{unit}}{A \vdash \textsf{if}\ (E)\ \textsf{then}\ S_1\ \textsf{else}\ S_2 : \textsf{unit}} \ (\textsc{If}) \qquad \frac{A \vdash E : \textsf{boolean} \quad A \vdash S : \textsf{unit}}{A \vdash \textsf{while}\ (E)\ S : \textsf{unit}} \ (\textsc{While})$$

Library Functions

The language allows programmers to invoke a the library functions listed below. These functions will be available in the library `libic.a` that we provide in the code generation assignment; the code generated by the compiler will be linked against this library.

```
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 */

static 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 i); /* returns a random number between 0 and n-1 */
int  time();        /* number of milliseconds since program start */
void exit(int i);   /* 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 will not be type-checked. For each qualified call `Library.foo`, the compiler will generate a static call to `__foo` in the assembly output, regardless of whether `foo` 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.

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{classDecl}^{*} \\
\mathit{classDecl} & ::= & \mathsf{class}\ \mathit{id}\ \ [\mathsf{extends}\ \mathit{id}]\ \ \texttt{'\{'}\ \ (\mathit{fieldDecl} \mid \mathit{methDecl})^{*}\ \ \texttt{'\}'} \\
 & & \\
\mathit{fieldDecl} & ::= & \mathit{type}\ \mathit{id}\ \ (\texttt{','}\ \ \mathit{id})^{*}\ \ \texttt{';'} \\
\mathit{methDecl} & ::= & (\mathit{type} \mid \mathsf{void})\ \ \mathit{id}\ \ \texttt{'('}\ [\mathit{formals}]\ \texttt{')'}\ \ \mathit{block} \\
\mathit{formals} & ::= & \mathit{type}\ \mathit{id}\ \ (\texttt{','}\ \ \mathit{type}\ \mathit{id})^{*} \\
 & & \\
\mathit{type} & ::= & \mathsf{int} \mid \mathsf{boolean} \mid \mathsf{string} \mid \mathit{id} \mid \mathit{type}\ \texttt{'['}\ \texttt{']'} \\
 & & \\
\mathit{block} & ::= & \texttt{'\{'}\ \mathit{varDecl}^{*}\ \ \mathit{stmt}^{*}\ \texttt{'\}'} \\
\mathit{varDecl} & ::= & \mathit{type}\ \ \mathit{id}\ [\texttt{'='}\ \mathit{expr}]\ \ (\texttt{','}\ \ \mathit{id}\ [\texttt{'='}\ \mathit{expr}])^{*}\ \ \texttt{';'} \\
 & & \\
\mathit{stmt} & ::= & \mathit{assign}\ \texttt{';'} \\
 & \mid & \mathit{call}\ \texttt{';'} \\
 & \mid & \mathsf{return}\ [\mathit{expr}]\ \ \texttt{';'} \\
 & \mid & \mathsf{if}\ \texttt{'('}\ \mathit{expr}\ \texttt{')'}\ \mathit{stmt}\ \ [\mathsf{else}\ \mathit{stmt}] \\
 & \mid & \mathsf{while}\ \texttt{'('}\ \mathit{expr}\ \texttt{')'}\ \mathit{stmt} \\
 & \mid & \mathsf{break}\ \texttt{';'} \mid \mathsf{continue}\ \texttt{';'} \\
 & \mid & \mathit{block} \\
 & & \\
\mathit{assign} & ::= & \mathit{location}\ \texttt{'='}\ \mathit{expr} \\
\mathit{location} & ::= & \mathit{id} \mid \mathit{expr}\ \texttt{'.'}\ \mathit{id} \mid \mathit{expr}\ \texttt{'['}\ \mathit{expr}\ \texttt{']'} \\
 & & \\
\mathit{call} & ::= & \mathit{method}\ \ \texttt{'('}\ [\mathit{actuals}]\ \texttt{')'} \\
\mathit{method} & ::= & \mathit{id} \mid \mathit{expr}\ \texttt{'.'}\ \mathit{id} \\
\mathit{actuals} & ::= & \mathit{expr}\ \ (\texttt{','}\ \ \mathit{expr})^{*} \\
 & & \\
\mathit{expr} & ::= & \mathit{location} \\
 & \mid & \mathit{call} \\
 & \mid & \mathsf{this} \\
 & \mid & \mathsf{new}\ \mathit{id}\ \texttt{'('}\ \texttt{')'} \\
 & \mid & \mathsf{new}\ \mathit{type}\ \texttt{'['}\ \mathit{expr}\ \texttt{']'} \\
 & \mid & \mathit{expr}\ \texttt{'.'}\ \mathsf{length} \\
 & \mid & \mathit{expr}\ \mathit{binary}\ \mathit{expr} \\
 & \mid & \mathit{unary}\ \mathit{expr} \\
 & \mid & \mathit{literal} \\
 & \mid & \texttt{'('}\ \mathit{expr}\ \texttt{')'} \\
 & & \\
\mathit{binary} & ::= & \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{'!='} \\
\mathit{unary} & ::= & \texttt{'-'} \mid \texttt{'!'} \\
\mathit{literal} & ::= & \mathit{integer\text{-}literal} \mid \mathit{string\text{-}literal} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null}
\end{array}
$$

Figure 1: IC Language Grammar