

The IC Language Specification

CS 0368-3133

Winter 2009–2010

Tel Aviv University*

The IC language is a simple object-oriented language that we will use in the compiler project. The goal is to build a complete compiler for this language and generate x86 assembly code. IC is essentially a subset of Java. It supports objects, subtyping, virtual and static methods, arrays; it is strongly-typed, and uses run-time checks to ensure the safety of object and array accesses; it supports dynamic allocation of objects and arrays, and uses an off-the-shelf garbage collector to reclaim heap memory. This document describes the structure and the semantics of IC.

1 Lexical Considerations

Identifiers and keywords are case-sensitive. Identifiers contain letters, digits, and the underscore character. Class identifiers start with an upper-case letter. All of the other identifiers start with a lower-case letter. The following are keywords and cannot be used as identifiers:

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

White spaces consist of space, tab, or newline characters. They 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.

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 are lexical errors.

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 $2^{31} - 1$.

*Course materials adapted with permission from Prof. Radu Rugina in Cornell university.

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 than 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`.

2 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:

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

Running the program with arguments `args` is equivalent to executing `A.main(args)`, where `A` is the class that contains the `main` method.

3 Variables

Program variables include local variables or method parameters, and are allocated on the run-time stack. Objects and arrays are allocated on the heap. Variables can hold integer and boolean values, or references to objects, arrays, or strings. Variables can be declared at any point in the program. They are visible from the declaration point up to the end of the innermost enclosing statement block.

Initialization. The program does not initialize variables with default values at declaration points. Instead, the compiler statically checks that each variable is assigned before it is used. Violations of this rule will cause compilation errors. On the other hand, object fields and array elements are initialized with default values (0 for integers, `false` for booleans, and `null` for references) when the objects or the arrays are created.

4 Strings

Unlike Java, IC has a primitive type `string`. Strings can be manipulated only in the following ways: using assignments of string references (including `null`); concatenating strings with the `+` operator; and testing for string reference equality using `==` (Note: this operator does not compare string contents). Library functions are provided to convert between strings and integer arrays containing the ascii codes of the characters in the string.

5 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 `new`, i.e., `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: $0 \leq i < n$. Violations will terminate the program with an appropriate error message.

6 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 A inherits all of the features (methods and fields) of class B . 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.

7 Subtyping

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

$$\frac{A \text{ extends } B \{ \dots \}}{A \leq B} \quad \frac{}{A \leq A} \quad \frac{A \leq B \quad B \leq C}{A \leq C} \quad \frac{}{\text{null} \leq A}$$

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.

8 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 the 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).

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 { int label; Node dest; }
```

9 Method Invocation

There are two kinds of method calls: static and virtual.

Static method calls are of the form $C.m(\dots)$, where C is a class name and m is a method declared static in C (or its superclasses). The program will execute this method, regardless of whether subclasses of C override this method or not. Hence, the method being executed is known at compile-time.

Virtual method calls are of the form $e.m(\dots)$, where e is an expression denoting an object reference. The method being executed is the method m of class C , where C is the *run-time* type of object e . Hence, the method executed is not known at compile-time. Virtual method calls are resolved at run-time via dynamic dispatch.

At each method invocation site, the program evaluates the actual arguments from left to right, and then assigns the computed values to the corresponding formal parameters of the method. Objects, arrays, and strings are passed by reference. The program then executes the body of the appropriate method, as described above. Upon return, the control is transferred

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. Furthermore, values returned by return statements must agree with the corresponding return types from method declarations. If a method is declared to return `void`, then return statements in the body of the method cannot return values. Such methods are allowed to reach the end of their body without a return statement. Otherwise, if the method is declared with a return type `T`, then all return statements must return values of type `T`. In this case, the method body is required to have a return statement on each program path. The compiler checks this requirement and reports an error whenever it is violated.

10 Scope rules

For each program, there is a hierarchy of scopes consisting of: the global scope, class scopes, method scopes, and local scopes for blocks within each method. The global scope consists of the names of all classes defined in the program. Each class has a static scope and an instance scope. The instance scope is the set of all fields and methods of that class; the static scope is the set of static methods only. 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 in that block.

When resolving an identifier at a certain point in the program, the enclosing scopes are searched for that identifier. In particular, local variables and method parameters can only be used after they are defined in one of the enclosing block or method scopes. Fields and methods, both static and virtual, can be used either directly (without the dot prefix) if the current class contains those fields or method. The current class scope is the instance scope if the current method is virtual; or the static class scope if the current method is static. Fields and virtual methods can be used in expressions of the form `e.f` or `e.m()` when `e` has class type `C` and the instance scope of `C` contains those fields and methods. Finally, static methods can be used in expressions of the form `C.m()` if the static scope of `C` contains `m`. Class names, fields, and methods can be used before the program declares them. However, the program must eventually declare them.

Identifiers, regardless of their kind, cannot be defined multiple times in the same scope. Here, *same* means exactly the same scope, not including the scopes it is nested in. The exception to this rule regards inheritance where identifiers, regardless of their kind, cannot be defined multiple times in the same scope or the scope of the superclass.

Therefore code like:

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

```
}
```

is illegal since 'foo' acts both as a field and a method name.

Code like this is legal:

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

Code like this is also legal:

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

Shadowing method parameters with local variables is illegal. For example:

```
class A {
    void foo(int x) {
        int x = 1;
    }
}
```

The following code is illegal since the same identifier (foo) appears both in the scope of the class A and in the scope of the class B, which inherits from A:

```
class A {
    int foo;
}
class B extends A {
    void foo() {}
}
```

11 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. In case 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 or 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.

12 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, Boolean, and **null** literals.

13 Operators

Unary and binary operators include the following:

- Arithmetic operators: addition `+`, subtraction `-`, multiplication `*`, division `/`, and modulo `%`. The operands must be integers. The plus operator `+` is also used to concatenate strings. 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	<code>[]</code> <code>()</code> <code>.</code>	array index, method call field/method access	left
2	<code>-</code> <code>!</code>	unary minus, logical negation	right
3	<code>*</code> <code>/</code> <code>%</code>	multiplication, division, remainder	left
4	<code>+</code> <code>-</code>	addition, subtraction	left
5	<code><</code> <code><=</code> <code>></code> <code>>=</code>	relational operators	left
6	<code>==</code> <code>!=</code>	equality comparison	left
7	<code>&&</code>	short-circuit and	left
8	<code> </code>	short-circuit or	left
9	<code>=</code>	assignment	right

14 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`, `CLASS`, `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.

```

program ::= classDecl*
classDecl ::= class CLASS [extends CLASS] '{' (field|method)*}'
  field ::= type ID (',' ID)* ';'
  method ::= [static] (type|void) ID '(' [formals] ')' '{' stmt*}'
  formals ::= type ID (',' type ID)*
  type ::= int | boolean | string | CLASS | type '['']'

  stmt ::= location '=' expr ';'
          | call ';'
          | return [expr] ';'
          | if '(' expr ')' stmt [else stmt]
          | while '(' expr ')' stmt
          | break ';'
          | continue ';'
          | '{' stmt*}'
          | type ID ['=' expr] ';'

  expr ::= location
          | call
          | this
          | new CLASS '(' ')'
          | new type '[' expr']'
          | expr '.' length
          | expr binop expr
          | unop expr
          | literal
          | '(' expr ')'

  call ::= staticCall | virtualCall
  staticCall ::= CLASS '.' ID '(' [expr (',' expr)*] ')'
  virtualCall ::= [expr '.' ] ID '(' [expr (',' expr)*] ')'
  location ::= ID | expr '.' ID | expr '[' expr']'

  binop ::= '+' | '-' | '*' | '/' | '%' | '&&' | '||'
          | '<' | '<=' | '>' | '>=' | '==' | '!='
  unop ::= '-' | '!'
  literal ::= INTEGER | STRING | true | false | null

```

Figure 1: IC Syntax

15 Typing rules

Typing rules for expressions

$$\begin{array}{c}
 \overline{E \vdash \text{true} : \text{bool}} \\
 \\
 \overline{E \vdash \text{integer-literal} : \text{int}} \\
 \\
 \frac{E \vdash e_0 : \text{int} \quad E \vdash e_1 : \text{int} \quad op \in \{+, -, /, *, \%\}}{E \vdash e_0 \text{ op } e_1 : \text{int}} \\
 \\
 \frac{E \vdash e_0 : T_0 \quad E \vdash e_1 : T_1 \quad T_0 \leq T_1 \text{ or } T_1 \leq T_0 \quad op \in \{==, !=\}}{E \vdash e_0 \text{ op } e_1 : \text{bool}} \\
 \\
 \frac{E \vdash e_0 : \text{bool} \quad E \vdash e_1 : \text{bool} \quad op \in \{\&\&, ||\}}{E \vdash e_0 \text{ op } e_1 : \text{bool}} \\
 \\
 \frac{E \vdash e_0 : T[] \quad E \vdash e_1 : \text{int}}{E \vdash e_0[e_1] : T} \\
 \\
 \frac{E \vdash e : T[]}{E \vdash e.\text{length} : \text{int}} \\
 \\
 \overline{E \vdash \text{new } T() : T} \\
 \\
 \frac{id : T \in E}{E \vdash id : T} \\
 \\
 \frac{E \vdash e_0 : T_1 \times \dots \times T_n \rightarrow T_r \quad E \vdash e_i : T'_i \quad T'_i \leq T_i \quad \text{for all } i = 1..n}{E \vdash e_0(e_1, \dots, e_n) : T_r}
 \end{array}
 \qquad
 \begin{array}{c}
 \overline{E \vdash \text{false} : \text{bool}} \\
 \\
 \overline{E \vdash \text{string-literal} : \text{string}} \\
 \\
 \frac{E \vdash e_0 : \text{string} \quad E \vdash e_1 : \text{string}}{E \vdash e_0 + e_1 : \text{string}} \\
 \\
 \frac{E \vdash e_0 : \text{int} \quad E \vdash e_1 : \text{int} \quad op \in \{<=, <, >=, >\}}{E \vdash e_0 \text{ op } e_1 : \text{bool}} \\
 \\
 \frac{E \vdash e : \text{int}}{E \vdash -e : \text{int}} \\
 \\
 \frac{E \vdash e : \text{bool}}{E \vdash !e : \text{bool}} \\
 \\
 \frac{E \vdash e : \text{int}}{E \vdash \text{new } T[e] : T[]} \\
 \\
 \overline{E \vdash \text{null} : \text{null}} \\
 \\
 \frac{E \vdash e : C \quad (id : T) \in C}{E \vdash e.id : T} \\
 \\
 \frac{(m : \text{static } T_1 \times \dots \times T_n \rightarrow T_r) \in C \quad E \vdash e_i : T'_i \quad T'_i \leq T_i \quad \text{for all } i = 1..n}{E \vdash C.m(e_1, \dots, e_n) : T_r}
 \end{array}$$

Typing rules for statements

$$\begin{array}{c}
\frac{E \vdash e_l : T \quad E \vdash e : T' \quad T' \leq T}{E \vdash e_l = e} \quad \frac{E \vdash e : \text{bool} \quad E \vdash S_1 \quad E \vdash S_2}{E \vdash \text{if } (e) S_1 \text{ else } S_2} \\
\\
\frac{E \vdash e : \text{bool} \quad E \vdash S_1}{E \vdash \text{if } (e) S_1} \quad \frac{E \vdash e : \text{bool} \quad E \vdash S}{E \vdash \text{while } (e) S} \quad \frac{}{E \vdash \text{break ;}} \quad \frac{}{E \vdash \text{continue ;}} \\
\\
\frac{E \vdash e_0(e_1, \dots, e_n) : T}{E \vdash e_0(e_1, \dots, e_n) ;} \text{ [CALL]} \quad \frac{E \vdash e : T \quad \text{ret} : T' \in E \quad T \leq T'}{E \vdash \text{return } e ;} \quad \frac{\text{ret} : \text{void} \in E}{E \vdash \text{return ;}} \\
\\
\frac{E \vdash e : T' \quad T' \leq T \quad E, x : T \vdash S}{E \vdash T x = e ; S} \text{ [DECL 1]} \quad \frac{E, x : T \vdash S}{E \vdash T x ; S} \text{ [DECL 2]} \quad \frac{S_1 \text{ not declaration} \quad E \vdash S_1 \quad E \vdash S_2}{E \vdash S_1 ; S_2}
\end{array}$$

Rule CALL says that a (virtual) method call, which is a legal expression is also legal as a statement. Rules DECL 1 and DECL 2 say that in order for a variable to be used in a statement it must be declared in a preceding statement.

Type-Checking Class and Method Declarations

$$\frac{\begin{array}{l} \text{classes}(P) = C_1, \dots, C_n \\ \vdash C_i \text{ for all } i = 1..n \end{array}}{\vdash P} \text{ [PROGRAM]}$$

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

$$\frac{E, x_1 : t_1, \dots, x_n : t_n, \text{ret} : t_r \vdash S_{\text{body}}}{E \vdash t_r m(t_1 x_1, \dots, t_n x_n) \{ S_{\text{body}} \}} \text{ [METHOD]}$$

Here, $\text{Env}(C, m)$ is the environment (or scope) for class C and method m . If m is virtual, $\text{Env}(C, m)$ is the instance scope of C and contains all of the methods and fields of C , including those declared in C 's superclasses. If m is static, $\text{Env}(C, m)$ is the static scope of C and contains only the static methods declared in C and its superclasses. Finally, $\text{classes}(C)$ yields all of the classes in program P ; and $\text{methods}(C)$ yields the declarations of all methods in the body of class C (but not those inherited from other classes).

16 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 in a file `libic.sig`. The compiler reads this file and uses the declared signatures to type-check any uses of library functions. In this file, library function signatures are declared inside a class `Library`. Currently, the following methods are supported:

```
class Library {
  static void println(string s); /* prints string s followed by a newline */
  static void print(string s);   /* prints string s */
  static void printi(int i);     /* prints integer i */
  static void printb(boolean b); /* prints boolean b */

  static int readi();           /* reads one character from the input */
  static string readln();      /* reads one line from the input */
  static 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 */
  static string itos(int i);     /* returns a string representation of i */
  static int[] stoa(string s);  /* an array with the ascii codes of chars in s */
  static string atos(int[] a);  /* builds a string from the ascii codes in a */

  static int random(int i);     /* returns a random number between 0 and n-1 */
  static int time();           /* number of milliseconds since program start */
  static void exit(int i);     /* terminates the program with exit code n */
}
```

All of the library methods are declared static, and are called accordingly, by qualifying the method call with the method name. For instance, `Library.random(100)` or `Library.stoi("412", -1)`. To generate the final executable, the assembly code generated by the compiler will be linked against library `libic.a`.

The grammar for the library signature file `libic.sig` is:

$$\begin{aligned} \textit{libic} &::= \textit{class Library} \{ \textit{libmethod}^* \} \\ \textit{libmethod} &::= \textit{static} (\textit{type} | \textit{void}) \textit{ID} '([\textit{formals}])' ';' \end{aligned}$$

The default location of the library signature file `libic.sig` is the current directory. Alternatively, the location of this file can be explicitly specified in the command line using an argument of the form `"-L /path/to/libic.sig"`.