Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2000 Handout 7 — Espresso Language Definition Wednesday, September 6

The project for the 18-unit flavor of the course is to write a compiler for a language called Espresso that supports object oriented design including data abstraction and inheritance via classes. We believe that Espresso is at least twice as time-consuming to implement as Decaf, the 12-unit language.

Example programs can be found online in `/mit/6.035/examples/espresso`.

Lexical Considerations

All Espresso keywords are lowercase. Keywords and identifiers are case-sensitive. For example, **if** is a keyword, but `IF` is a variable name; `foo` and `Foo` are two different names referring to two distinct variables.

The reserved words are:

> **boolean callout class else extends false if int new null return this true void while**

Comments are started by `//` and are terminated by the end of the line.

White space may appear between any lexical tokens. White space is defined as one or more spaces, tabs, page-breaking and line-breaking characters, and comments.

Keywords and identifiers must be separated by white space, or a token that is neither a keyword nor an identifier. `thiswhiletrue` is a single identifier, not three distinct keywords.

String literals are composed of ⟨char⟩s enclosed in double quotes. A character literal consists of a ⟨char⟩ enclosed in single quotes.

Numbers in Espresso are 32-bit signed. That is, between decimal values -2147483648 to 2147483647.

A ⟨char⟩ is any printable ASCII character (ASCII values between decimal value 32 and 126, or octal 40 and 176) other than quote (`"`), single quote (`'`), or backslash (\), plus the 2-character sequences "`\"`" to denote quote, "`\'`" to denote single quote, "`\\`" to denote backslash, "`\t`" to denote a literal tab, or "`\n`" to denote newline.

Reference Grammar

Meta-notation:

$\langle$foo$\rangle$	means foo is a nonterminal.
foo	(in **bold** font) means that **foo** is a terminal i.e., a token or a part of a token.
$\left[x\right]$	means zero or one occurrence of x, *i.e.,* x is optional; note that brackets in quotes '[' ']' are terminals.
x^*	means zero or more occurrences of x.
x^+**,**	a comma-separated list of one or more x's.
$\left\{ \right.$ $\left. \right\}$	large braces are used for grouping; note that braces in quotes '{' '}' are terminals.
$\mid$	separates alternatives.

⟨program⟩ → ⟨class_decl⟩$^+$

⟨class_decl⟩ → **class** ⟨id⟩ $\left[$**extends** ⟨id⟩$\right]$ '{' ⟨var_decl⟩* ⟨method_decl⟩* '}'

⟨optional_int_size⟩ → $\left[$⟨int_literal⟩$\right]$

⟨method_decl⟩ → $\left\{$⟨type⟩ | **void**$\right\}$ ⟨id⟩ ($\left[\left\{$⟨type⟩ ⟨id⟩ | ⟨type⟩ ⟨id⟩ '[' ']' $\right\}^+$,$\right]$) ⟨block⟩

⟨block⟩ → '{' ⟨var_decl⟩* ⟨statement⟩* '}'

⟨var_decl⟩ → ⟨type⟩ $\left\{$⟨id⟩ | ⟨id⟩ '[' ⟨optional_int_size⟩ ']' $\right\}^+$, ;

⟨type⟩ → **int** | **boolean** | ⟨id⟩

⟨statement⟩ → ⟨location⟩ = ⟨expr⟩ ;
| ⟨method_call⟩ ;
| **if** (⟨expr⟩) ⟨block⟩ $\left[$**else** ⟨block⟩$\right]$
| **while** (⟨expr⟩) ⟨block⟩
| **return** $\left[$⟨expr⟩$\right]$;
| ⟨block⟩

⟨method_call⟩ → $\left[\left\{$⟨simple_expr⟩ | ⟨id⟩$\right\}$.$\right]$ ⟨method_name⟩ ($\left[$⟨expr⟩$^+$,$\right]$)
| **callout** (⟨string_literal⟩ $\left[$, ⟨callout_arg⟩$^+$,$\right]$)

⟨method_name⟩ → ⟨id⟩

⟨location⟩ → ⟨id⟩
| ⟨simple_expr⟩ '[' ⟨expr⟩ ']'
| ⟨simple_expr⟩ . ⟨id⟩

⟨expr⟩ → ⟨simple_expr⟩
| **new** ⟨id⟩ `( )`
| **new** ⟨type⟩ `[` ⟨expr⟩ `]`
| ⟨literal⟩
| ⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩
| `-` ⟨expr⟩
| `(` ⟨expr⟩ `)`

⟨simple_expr⟩ → ⟨location⟩
| **this**
| ⟨method_call⟩

⟨callout_arg⟩ → ⟨expr⟩ | ⟨string_literal⟩

⟨bin_op⟩ → ⟨arith_op⟩ | ⟨rel_op⟩ | ⟨eq_op⟩ | ⟨cond_op⟩

⟨arith_op⟩ → `+` | `-` | `*`

⟨rel_op⟩ → `<` | `>` | `<=` | `>=`

⟨eq_op⟩ → `==` | `!=`

⟨cond_op⟩ → `&&` | `||`

⟨literal⟩ → ⟨int_literal⟩ | ⟨char_literal⟩ | ⟨bool_literal⟩ | **null**

⟨id⟩ → ⟨alpha⟩ ⟨alpha_num⟩*

⟨alpha_num⟩ → ⟨alpha⟩ | ⟨digit⟩

⟨alpha⟩ → `a` | `b` | `...` | `z` | `A` | `B` | `...` | `Z` | `_`

⟨digit⟩ → `0` | `1` | `2` | `...` | `9`

⟨hex_digit⟩ → ⟨digit⟩ | `a` | `b` | `c` | `d` | `e` | `f` | `A` | `B` | `C` | `D` | `E` | `F`

⟨int_literal⟩ → ⟨decimal_literal⟩ | ⟨hex_literal⟩

⟨decimal_literal⟩ → [`-`] ⟨digit⟩ ⟨digit⟩*

⟨hex_literal⟩ → `0x` ⟨hex_digit⟩ ⟨hex_digit⟩*

⟨bool_literal⟩ → **true** | **false**

⟨char_literal⟩ → `'` ⟨char⟩ `'`

⟨string_literal⟩ → `"` ⟨char⟩* `"`

Semantics

Espresso is a strongly-typed, object-oriented language with support for both inheritance and encapsulation. Its syntax and semantics are essentially a subset of those of Java. There are two basic types in Espresso: integers (int) and booleans (boolean.) In addition, programmers can define their own object types using classes. Finally, there is a built-in array type: T[] is an array of elements of type T, where T can be either a basic type or a class.

A Espresso program consists of a series of one or more class declarations. The program should contain a declaration for a class called **Program** with a method called **main** that has no parameters. Execution of a Espresso program starts at method **main**.

A user-defined object is an instance of a user-defined class, and consists of a set of named fields and methods. Fields are a set of private variables attached to an object that maintain the internal representation for the object. Methods define the public interface to an object type and can be called by anyone.

Espresso enforces object encapsulation: other object classes can only manipulate an object by invoking methods on it and are not able to directly access or modify the internal representation in the fields.

Espresso supports inheritance, allowing a derived class to extend a base class by adding additional fields and methods and by overriding existing methods with new definitions. Espresso supports automatic "up-casting" so that an object of a derived class type can be provided whenever an object of its base class type is expected. However, Espresso does not permit overloading *i.e.,* the use of the same name for methods with different signatures.

Locations

Espresso has three kinds of locations: variables (i.e. identifiers), array elements, and object fields. Each location has a type. Locations of types **int** and **boolean** contain integer values and boolean values, respectively. Locations of other types denote objects or arrays. Such locations contain pointers to objects or arrays in the heap.

Each location is initialized to a default value when it is declared. Array and object locations have a default value of **null**, integers have default value of zero, and booleans have a default value of false. Object fields are initialized when an object is constructed using new. Local variables are initialized on entry to a method. Array elements are initialized when an array is created.

A location can be assigned **null** even after it has been initialized.

Array Types

Espresso allows programmers to create and use one-dimensional arrays of any element type, except that an array element cannot itself be an array (however, an array element can be an object that contains an array).

Arrays are indexed from 0 to $N - 1$, where N is the size of the array. The usual bracket notation is used to index arrays.

Arrays are created using the new operator. **new** ⟨type⟩[N] allocates a new array of type ⟨type⟩ and size N, where N must be strictly positive.

Arrays support the indexing operation [] to get or set a given element in the array and the **length** field for returning the length of the array.

If ⟨id⟩ is an array of length N, ⟨id⟩.**length** returns N and ⟨id⟩[i] returns a reference to the i^{th} element of ⟨id⟩, unless $i < 0$ or $i \geq N$ or ⟨id⟩ is set to **null**, in which case a runtime error results.

Inheritance and Subtyping

Espresso supports inheritance using the construction *T2* **extends** *T1* in the class declaration, which means that the *derived class* T2 inherits all of the fields and methods of the *base class* T1, in addition to any that it declares on its own.

We say that a class S is *derived* from a class T if it either extends T or extends a class which is derived from T. In other words, "is derived from" is the transitive closure of "extends".

T2 can *override* existing methods in a class it is derived from by defining a new method with the same name, argument types, and return types as the method in the base class. This allows T2 to reuse code by inheriting methods and overriding the the methods whose behavior it needs to change.

Because a T2 supports all of the same methods as a T1, it can be used anywhere a T1 is expected (for example, as a method argument); this is called subtyping or polymorphism. Note that if T2 overrides a method m in T1, the overriding method will be called even when an object of type T2 is stored in a location of type T1.

For example, suppose a base class called SimpleAlert class defines a beep() method to alert the user by making a sound. Using inheritance, we can define a derived class VisualAlert that overrides beep() to instead flash the screen. A VisualAlert can be used anywhere an Alert is expected, and whenever the beep method is invoked such an object, the visual version that flashes the screen will be called.

Overriding means that the compiler cannot statically determine the exact address of the method that should be called when the user invokes a method on an object — it could be the either method defined in the declared type of the object or an overriding method declared in any of the classes derived from that type. The method to be called is therefore determined at runtime by consulting a table associated with each object containing pointers to the methods for that object. This table is called a method dispatch table; there is a distinct dispatch table for each class. We will discuss dispatch tables in more detail in a later handout.

We define a type T as *compatible* with another type S if $T = S$ or if T is derived from S. Note that compatibility is not symmetric: if `banana` is compatible with `fruit`, this does not imply `fruit` is compatible with `banana`. In general, if T is compatible with S, an object of type T can be used anywhere an object type S is expected.

There is no subtyping of array types: if T2 extends T1, an array of T2[] is *not* compatible with an array of T1[], in contrast to Java.

Scope Rules

Espresso has simple scope rules. All identifiers must be defined (textually) before use. For example:

- a variable must be declared before it is used.
- a method can be called only by code appearing after its header.
- a class can only be used in a declaration appearing after the name of the class is first defined.

When resolving an identifier in a Espresso method, at least three scopes are considered: the global scope, the object scope, the method scope, and any local scopes defined within a particular method.

The global scope includes the names of all the classes that have been defined. The object scope includes all of the fields and methods defined in an object's class (including of course inherited members.) The method scope includes all of a method's formal parameters, as well as the variables defined in the method. Local scopes exist within any ⟨block⟩s defined in any code. Inner scopes shadow outer scopes; a variable defined in the method scope masks another variable with the same name in the object scope, for example. Similarly, variables defined in local scopes shadow variables in less-deeply nested local scopes and the method and object scopes.

The global scope is used only in contexts where a class name is expected; there is no context in which an identifier may be drawn either from the class scope or another scope.

No identifier may be defined more than once in the same scope. Thus class names must all be distinct in the global scope. Similarly, field and method names must all be distinct in the object scope, and local variable names and formal parameters names must be distinct in the method scope.

Scoping for location and method identifiers When resolving an identifier in an expression of the form ⟨id⟩ or ⟨id⟩ `'['` ⟨expr⟩ `']'`, the compiler first considers the most-deeply nested local scope. If ⟨id⟩ is the name of a local variable, the location refers to the value of that variable.

If ⟨id⟩ is the pseudo-local **this**, the location refers to the current object (the object on which the current method was invoked.)

Otherwise, the compiler considers less-deeply nested scopes in turn. If ⟨id⟩ is the name of a variable in that scope, the location refers to the value of that variable. Failing this, the compiler then considers ⟨id⟩ in the method and object scopes.

If ⟨id⟩ is found in none of these scopes, a compile-time error results.

For locations of the form ⟨simple_expr⟩ `.` ⟨id⟩, ⟨id⟩ is resolved against the object scope for ⟨simple_expr⟩. The type of ⟨simple_expr⟩ must be a class type T, ⟨id⟩ must be name one of T's fields, and the location must appear textually in a method of type T.

For method calls of the form $\left[\langle\text{simple\_expr}\rangle\texttt{.}\right]\langle\text{id}\rangle$ `()`, ⟨id⟩ is resolved against the object scope for ⟨simple_expr⟩, or that of the current object if the the expression is omitted. The type of ⟨simple_expr⟩ must be a class type T, and ⟨id⟩ must be name one of T's methods.

Note that classes cannot access fields of other, unrelated classes: methods of a class can only access fields on an object of the same class or a derived class.

For example, if a class Point includes a field `x`, a method of Point (or any class derived from point) can reference that field simply with `x`, unless there is a local variable called `x`, in which case it can reference this field using **this.x**.

Assignment

For the types **int** and **boolean**, Espresso uses value-copy semantics; the assignment ⟨location⟩ `=` ⟨expr⟩ copies the value resulting from the evaluation of ⟨expr⟩ into ⟨location⟩.

For arrays and objects, Espresso uses reference-copy semantics; the assignment ⟨location⟩ `=` ⟨expr⟩ causes ⟨location⟩ to contain a reference to the object resulting from the evaluation of ⟨expr⟩ (*i.e.*, the assignment copies a pointer to an object rather than the object.)

The ⟨location⟩ and the ⟨expr⟩ in an assignment must either have the same type, or they must both be classes such that the type of ⟨location⟩ is an ancestor of the type of ⟨expr⟩ in the inheritance hierarchy.

It is legal to assign to a formal parameter variable within a method body. Such assignments affect only the method scope.

Method Invocation and Return

Method invocation involves (1) passing argument values from the caller to the callee, (2) executing the body of the callee, and (3) returning to the caller, possibly with a result.

Argument passing is defined in terms of assignment: the formal arguments of a method are considered to be like local variables of the method and are initialized, by assignment, to the values resulting from the evaluation of the argument expressions. The arguments are evaluated from left to right.

The body of the callee is executed by then executing the statements of its method body.

A method that has a declared result type of **void** can only be called as a statement, *i.e.*, it cannot be used in an expression. Such a method returns control to the caller when **return** is called (no result expression is allowed) or when the textual end of the callee is reached.

A method that returns a result may be called as part of an expression, in which case the result of the call is the result of evaluating the expression in the **return** statement when this statement is reached. It is illegal for control to reach the textual end of a method that returns a results.

A method that returns a result may also be called as a statement. In this case, the result is ignored.

Control Statements

The **if** statement has the usual semantics. First, the ⟨expr⟩ is evaluated. If the result is **true**, the **then** arm is executed. Otherwise, the **else** arm is executed, if it exists. Since Espresso requires that the **then** and **else** arms be enclosed in braces, there is no ambiguity in matching an **else** arm with its corresponding **if** statement.

The **while** statement has the usual semantics. First, the ⟨expr⟩ is evaluated. If the result is **false**, control exits the loop. Otherwise, the loop body is executed. If control reaches the end of the loop body, the **while** statement is executed again.

Expressions

Expressions follow the normal rules for evaluation. In the absence of other constraints, operators with the same precedence are evaluated from left to right. Parentheses may be used to override normal precedence.

A location expression evaluates to the value contained by the location.

Method invocation expressions are discussed in *Method Invocation and Return*. **new** expressions are discussed in *Abstract Types*. I/O related expressions are discussed in *Library Callouts*.

Integer literals evaluate to their integer value. Character literals evaluate to their integer ASCII values, *e.g.,* `'A'` represents the integer 65. (The type of a character literal is **int**.)

The arithmetic operators (⟨arith_op⟩ and unary minus) have their usual precedence and meaning, as do the relational operators (⟨rel_op⟩). Relational operators are used to compare integer expressions.

The equality operators `==` and `!=` can be used to compare any two expressions having the same type. (`==` is "equal" and `!=` is "not equal"). For types **int** and **boolean**, value equality is used. For all other types, reference equality is used.

The boolean connectives `&&` and `||` are interpreted using short circuit evaluation as in Java. The side-effects of the second operand are not executed if the result of the first operand determines the value of the whole expression (i.e., if the result is false for `&&` or true for `||`).

Operator precedence, from highest to lowest:

Operators	*Comments*
`.`	field selection
`-`	unary minus
`*`	multiplication
`+ -`	addition, subtraction
`< <= >= >`	relational
`== !=`	equality
`&&`	conditional and
`||`	conditional or

Library Callouts

Espresso includes a primitive method for calling functions provided in the runtime system, such as the standard C library or user-defined functions.

The primitive method for calling functions is:

> **int callout** (⟨string_literal⟩, $\left[\langle \text{callout\_arg} \rangle^{+}\,\textbf{,}\right]$) — the function named by the initial string literal is called and the arguments supplied are passed to it. Expressions of boolean or integer type are passed as integers; string literals or expressions with array type are passed as pointers. The return value of the function is passed back as an integer. The user of **callout** is responsible for ensuring that the arguments given match the signature of the function, and that the return value is only used if the underlying library function actually returns a value of appropriate type. Arguments are passed to the function in the system's standard calling convention.

In addition to accessing the standard C library using **callout**, an I/O function can be written in C (or any other language), compiled using standard tools, linked with the runtime system, and accessed by the **callout** mechanism.

Semantic Rules

These rules place additional constraints on the set of valid Espresso programs. The behavior of a program that violates these rules is undefined. While explicitly checking each rule is not required, a robust compiler will avoid crashing when presented with an invalid program.

1. No identifier is declared twice in the same scope.
2. No identifier is used before it is declared (in an appropriate scope).
3. The program should contain a definition for a class called **Program** with a method called **main** that has no parameters. (Note that since execution starts at method **main**, any methods defined after it will never be executed.)
4. The number of arguments in a method call must be the same as the number of formals, and the types of the arguments in a method call must be compatible with the types of the formals.
5. If a method call is used as an expression, the method must return a result.
6. A **return** statement must not have a return value unless it appears in the body of a method that is declared to return a value.
7. The expression in a **return** statement must have the result type that is compatible with the enclosing method definition.
8. For all method invocations of the form ⟨simple_expr⟩ `.` ⟨id⟩ `()`
   (a) the type of ⟨simple_expr⟩ must be some class type, T, and
   (b) ⟨id⟩ must name one of T's method
9. An ⟨id⟩ used as a ⟨location⟩ must name a declared variable or field.
10. For all locations of the form ⟨simple_expr⟩ `[`⟨expr⟩`]`
    (a) the type of ⟨simple_expr⟩ must be an instantiation of **array**, and
    (b) the type of ⟨expr⟩ must be **int**.
11. For all locations of the form ⟨simple_expr⟩ `.` ⟨id⟩
    (a) the type of ⟨simple_expr⟩ must be some class type, T, or ⟨simple_expr⟩ must be `this` (in which case its type is that of the enclosing class.)
    (b) ⟨id⟩ must name one of T's fields, and
    (c) the location must appear textually in an method of type T.
12. The ⟨expr⟩ in **if** and **while** statements must have type **boolean**.
13. The operands of ⟨arith_op⟩s and ⟨rel_op⟩s must have type **int**.
14. The operands of ⟨eq_op⟩s must have compatible types (i.e. either the first operand is compatible with the second or vice versa.).
15. The operands of ⟨cond_op⟩s must have type **boolean**.
16. In calls to **new** `T()`, T must be the name of a class previously defined in the global scope.
17. In calls to **new** `T[size]`, size must be an integer, and T must be either **int**, **boolean**, or the name of a class in the global scope. Note that T can *not* be an array type.
18. A field of a class T1 can only be accessed outside an operation for T1 if the caller is an operation in class T2 such that T2 is derived from T1.

19. The **extends** construct must name a previously-declared class; a class cannot extend itself.

20. The same name cannot be used for a field in two separate classes when one class is an ancestor of the other in the inheritance graph.

21. The type of the ⟨expr⟩ in an assignment must be compatible with the type of the ⟨location⟩.

22. If T2 is derived from T1 and ⟨id⟩ is a method name defined in both T1 and T2, then m must have exactly the same signature and T1 and T2.

Run Time Checking

1. The subscript of an array must be in bounds. If `a` is an array of size `N`, the index must be in the range term 0 ... (`a.length`-1).

2. In calls to **new** `T[size]`, size must be non-negative.

3. Control must not fall off the end of a method that is declared to return a result.

4. The value of a location must be non-**null** whenever it is dereferenced using `.` or [].

When a run-time errors occurs, an appropriate error message is output to the terminal and the program terminates. Such error messages should be helpful to the programmer trying to find the problem in the source program.