The Eck Programming Language

CS420 Compilers and Translators
Spring 2020

Background

The Eck programming language is a simple, object-based language with a syntax and semantics similar to Java, but with no support for inheritance. It is not intended to be a "real" programming language suitable for writing large programs; instead, it is designed as a vehicle for exploring the creation of compilers at the undergraduate level. Eck is based on the Jack language used in Nisan and Schocken's popular "Nand to Tetris" course, with many of Jack's syntactic and semantic over-simplifications replaced with features typically found in "real" languages in order to provide a suitable challenge for upper-class computer science majors.

Syntax

In the syntax descriptions below, the "*" signifies zero or more instances and the "?" signifies zero or one instance.

Classes

An Eck program is a collection of classes, each appearing in a separate file. Class declarations have the following format:

<classDec> → **class** <className> **{** <classVarDec>* <subroutineDec>* **}**

<classVarDec > → (**static** | **field**) <type> <varName> (**,** <varName>)? **;**

<type> → **int** | **char** | **boolean** | <className> |
int[] | **char[]** | **boolean[]**

< subroutineDec > → (**constructor** | **function** | **method**) (**void** | <type>)
<subroutineName> **(**<parameterList>**)** **{** <subroutineBody> **}**

< parameterList > → ((<type> <varName>) (**,**<type> <varName>)*) ?

<subroutineBody> → <varDec>* <statement>*

<varDec> → <type> <varName> (**,** <varName>)* **;**

<className> → *identifier*
<subroutineName> → *identifier*
<varName> → *identifier*

Each class has a name through which the class can be globally accessed. Then comes a sequence of zero or more declarations for *static* (i.e., class) or *field* (i.e., instance) variables. Finally, there are zero or more subroutine declarations.

Subroutines

Subroutines can be either:

- *Constructors*, which create an instance of the class.
- *Functions*, analogous to the static functions in Java, which are associated with the class itself.
- *Methods*, which are associated with instances of the class.

Each subroutine has a *name* though which it is accessed, and a *type* describing the value returned by the subroutine. If the subroutine does not return a value, it is declared to be of type *void*. Constructors can have arbitrary names, but they must return an object of the class type.

Following its specifier, return type, and name, the subroutine declaration contains a sequence of zero or more local variable declarations, then a sequence of zero or more statements. Subroutines in Eck can be *overloaded*, meaning that a class may contain more than one subroutine definition of the same name, if the type signature of each definition is different. A subroutine declaration does not have to appear earlier in the code than any calls to the subroutine.

Within a class, methods are called using the syntax <methodName> **(**<argumentList>**)**, while functions and constructors must be called using their full name:

<className> **.** <subroutineName> **(**<argumentList>**)**

Outside a class, the class functions and constructors are also called using their full names, while methods are called using the syntax <varName> **.** <methodName> **(**<argumentList>**)** where <varName> is a previously defined object variable.

Subroutine parameters are passed by value.

An Eck program is a collection of one or more classes that reside in the same directory. One class must be named `Main`, and this class must include a function named `main()`. When an Eck program is executed, the Eck runtime environment will automatically start running the `Main.main()` function.

Statements

Eck has five generic statements, described by the following rules:

<statement > → <assignmentStatement> |<ifStatement> | <whileStatement> | <doStatement> |<returnStatement>

< assignmentStatement > → <varName> (**[** <expression> **]**)? **=** <expression> **;**

<ifStatement> → **if** **(** <expression> **)** **{** <statement>* **}** (**else** **{** <statement>* **}**)?

< whileStatement > → **while** **(** <expression> **)** **{** <statement>* **}**

< doStatement > → **do** <subroutineCall>

< returnStatement > → **return** < expression>? **;**

Statements are executed for their side effects; they do not evaluate to a value.

Expressions

An Eck expression is one of the following:

<expression> → <term> (<binaryOp> <term>)*

<term> → *integerConstant* | *stringConstant* | <keywordConstant> | <varName> | <varName> **[** <expression>**]** | <subroutineCall> | **(** <expression> **)** | <unaryOp> <term>

<subroutineCall> → <subroutineName> **(** <expressionList> **)** | (<className> | <varName>) **.** <subroutineName> **(** <expressionList> **)**

<expressionList> → (<expression> (**,** <expression>)*)?

<binaryOp> → **+** | **-** | ***** | **/** | **&** | **|** | **<** | **>** | **=**

<unaryOp> → **-** | **~**

<keywordConstant> → **true** | **false** | **null** | **this**

All binary operators are left associative. The order of precedence for the binary operators is, from highest to lowest:

```
*, /
+, -
<, >, =
&, |
```

The unary operators have higher precedence than the binary operators.

Types

Eck is a statically typed language, meaning the type of every expression can be determined at compile time. It is also a strongly typed language, meaning that all possible type errors can be discovered at compile time.

Eck has three *primitive types*:

- `int`, which is a 16-bit 2's complement integer.
- `boolean`, which consists of the values `true` and `false`. Internally, Boolean variables are represented as 16-bit integers, with `true` represented by 1 and `false` represented by 0.
- `char`, which is a Unicode character represented internally as a 16-bit integer.

Variables

There are four kinds of variable in Eck:

- *Static* variables are defined at the class level and are shared by all objects derived from the class. Static variables are private to the class and cannot be directly accessed from outside the class.

- *Field* variables define the properties of individual instances of the class. Each instance object gets its own set of field variables. Field variables are private to the class and cannot be directly accessed from outside the class.
- *Local* variables are declared in subroutines and exist only as long as the subroutine is running.
- *Parameter* variables are used to pass arguments to subroutines. You can think of them as local variables that get assigned to the actual parameter values at the time a subroutine is called.

Variables in Eck must be declared before they are used. Each variable in Eck is either a primitive type, an array of primitive type, or an *object type* which has the name of a class. The class that implements an object type can be either part of the Eck standard library (e.g., `String`) or it may be any user-defined class residing in the program directory.

Variables of primitive type are allocated to memory when declared, and given the initial value of zero.

Object Construction

At declaration time, variables of object type are allocated a memory location containing a reference to an object. That reference will initially be `null`; it will be assigned an actual value later, if and when the programmer actually constructs an object by calling a constructor for the object's class and assigning the result to the variable.

Arrays

Arrays can only be of primitive type. Arrays are one-dimensional and the first index is always zero. Access to array elements is done using the typical `a[j]` notation.

Arrays are implemented using the `Array` class that is part of the Eck standard library, so the compiler should treat an array the same way it treats objects. The programmer is responsible for instantiating and disposing of the array.

Strings

Strings are declared using a built-in class called `String`. The Eck compiler recognizes the syntax `"abc"` and treats it as the contents of some `String` object. The contents of `String` objects can be accessed and modified using the methods of the `String` class, as documented in its API. The compiler is responsible for constructing `String` objects for any string constants that appear in a program by first using the `String.new(`*`int maxLength`*`)` constructor to construct a new empty string (of length zero) that can contain at most *maxLength* characters, and then repeatedly calling `String.appendChar(char c)` to build up the string constant character by character.

Type Conversions

If a program attempts to perform an arithmetic operation on a character, the Eck compiler will first convert the Unicode character into its integer representation. Likewise, if an integer value is used where a character value is expected, the compiler will convert the integer into a Unicode character. No conversions to or from `boolean` is allowed.