The Easy Language

José de Oliveira Guimarães
Departamento de Computação
UFSCar - São Carlos, SP
Brasil
e-mail: jose@dc.ufscar.br

March 19, 2004

1 Introduction

Easy is a very simple and *easy* to learn programming language featuring modules and basic procedural constructs. An example of a program in Easy is shown in Figure 1. The reader will immediately note that semicolons (`;`) are not used in the language.

1.1 Modules

A program in Easy is composed by one or more modules. A module is composed by an import list, an export list, a list of module constants, a list of module variables, and a list of routines, in this order. One or more lists may be absent. All are optional. A module may export constants, variables, and routines. The exported items are public — they can be used by other modules that import the module. In the example, module `main` imports module `Math` and therefore it can use routine `factorial` in

```
sum = sum + Math.factorial(value)
```

The module name should always precede the identifier name.

Routine `factorial` could be used because it was exported by module `Math`:

```
module Math
   export factorial
   ...
end Math
```

The keyword `end` that ends a module must be followed by the module name. Of course, a module only imports modules defined textually before it.

Every program must have a module called `main` with a parameterless routine called `run` that returns nothing. The program execution starts at `run`, which must be exported by module `main`.

1.2 Comments

Anything between `{` and `}` is a comment. Nested comments are not allowed.

1.3 Types

There are only three types in Easy: `int`, `bool`, and `String`. Type `int` represents integers and its literals must be between 0 and 32767. Any number of zeros before the number is allowed. The `bool` type has only two values: `true` and `false`.

```
  { This is a comment }
module Math
   export factorial
      { factorial is a function taking an integer as parameter and returning
        an integer. It is exported and can be used by other modules }

   routine factorial( int n ) -> int
      is
         if n > 1
            return n*factorial(n-1)
         else
            return 1
      end factorial
end Math

module main
   import Math
   export run
   const int Max = 10       { Max is a module constant }
   var int sum              { sum is a module variable }
     { f is a procedure taking no parameters. It has a local variable n and calls
       function factorial of module Math }
   routine f
      local int n   { declaration of a local variable }
      is
            { getInt returns an integer read from the standard input }
         n = In.getInt
         if (n > Max) or n < 1
            begin
            Out.writeln("Number out of limits: ", n)
            return
            end
         sum = 0
         loop
            value = In.getInt
            sum = sum + Math.factorial(value)
            n = n - 1
               { when n <= 0, the loop ends }
            exiton n <= 0
         endloop
         Out.writeln(sum)
      end f

   routine run   is   f   end run
end main
```

Figure 1: An example in Easy

Unless stated otherwise, the arithmetical, comparison, and logical operators follow the rules of language C. The comparison operators may be applied to the types `int` and `bool`. Consider that `true` > `false`. Of course, both operands should belong to the same type. The resultant value has type `bool`.

A `String` literal is anything between double quotes (`"`):

```
"Hello !!!"
"He said 'hello !!!'"
```

The arithmetical operators take integers and return integers. The comparison operators always return values of type `bool`.

1.4 Identifiers

Identifiers are composed by any number of letters, digits, and underscore (_). They should have at least one letter. The following identifiers are valid:

```
count, 2nd, result, 0123456789_ten  012w
```

Note that an identifier can begin with a number. Upper and lower case letters in identifiers are considered equal. However, the language keywords should always be in lower case.

1.5 Scope

The scope of a module constant, variable, or routine is the place of declaration to the end of the module. Of course, an imported identifier can be used outside its original module, like `factorial` in the example. It should always be preceded by the module name. Only public (the exported) identifiers can be used outside its module.

The scope of local variables and parameters is the whole routine. A local variable cannot have the same name as a parameter. A module variable cannot have the same name as a module constant. A routine cannot have the same name as a module variable, constant, or routine. A local variable or a parameter can have the same name as a module variable or constant. However, it would hide the module identifier. Inside a module `M` with identifier `I`, one cannot access `I` using "`M.I`". The code below is illegal.

```
module M
   export f
   var int I
   routine f
      local bool I
      is
         I = false
         M.I = 0    { compile error !!! }
         ...
      end f
end M
```

1.6 Assignments

An assignment

```
v = expr
```

is valid if the type of `v` is the same as the type of `expr`. The same rules apply to parameter passing and return of values in functions.

1.7 Routine Call

A parameterless routine should be called without parentheses:

```
process
v = calculate
```

Assume `process` is a routine taking no parameters and returning nothing and `calculate` is a parameterless routine returning an `int`. If the routine has one or more parameters, we should use `()`:

```
v = factorial(n)
printPersonData("Marcia", "Penapolis", 27)
```

The real parameters should have the same types as the corresponding formal parameters (those declared in the routine). The return value of a function is given by the `return` command:

```
return expr
```

The `expr` type should be the same as the function type. A `return` may appear in a routine without return type. It should not take any parameters.

1.8 Decision Statements

The syntax of the `if` command is

```
if  Expression
   Statement
[ else Statement ]
```

in which [and] denote an optional item. Note that there is no `then`. `Expression` should have type `bool`.

1.9 Loop Statements

There is only one loop statement:

```
LoopStatement ::= "loop"  StatementList  "endloop"
```

The statements are repeated till the expression in one `exiton` statement becomes true. For example, the loop

```
i = 0
sum = 0
loop
   sum = sum + i
   i = i + 1
   exiton i >= 10
endloop
```

repeats from `i = 0` to `i = 10`. Statement `exiton` may appear anywhere inside the `statement`. It always refers to the inner loop statement:

```
...
loop  { loop 1 }
   sum = 0
   loop  { loop 2 }
      ...
      exiton n < 0   { refer to loop 2 }
```

```
      ...
   endloop
   exiton sum > Max  { refer to loop 1 }
   sum = sum + n
endloop
```

Of course, `exiton` can only appear inside a `loop` command.

1.10 Input and Output

Input from the standard input is made by the `getInt` and `getString` functions of the predefined module `In`:

```
n = In.getInt
name = In.getString
```

Of course, `getInt` returns an `int` and `getString` returns a `String`.

Output to the standard output is made by the `write` and `writeln` routines of the predefined module `Out`:

```
Out.writeln("The sum is ", sum)
```

Any number of `int` and `String` values can be parameters to these routines. `write` prints all its parameters separated by a white space. `writeln` does the same and at the end it prints a newline character (it may be CR/LF in some machines).

2 The Easy Grammar

This section defines the Easy grammar. The reserved words and language symbols are quoted. Any sequence of symbols between { and } can be repeated zero or more times. Any sequence of symbols between [and] is optional. Parentheses group symbols. For example,

D ::= (A|B) { C }

means A or B followed by any number of C's.

The rules IntValue and StringValue represent integer and string literals. Id represents identifiers. Note that the input and output routines of modules are not in the grammar — they belong to the predefined modules `In` and `Out`.

Program	::= Module { Module }
Module	::= "module" Id [ImportList] [ExportList] [ModuleConstDecList] [ModuleVarDecList] [RoutineList] "end" Id
ImportList	::= "import" IdList
IdList	::= Id { "," Id }
ExportList	::= "export" IdList
ModuleConstDecList	::= "const" ConstDec { "," ConstDec }
ConstDec	::= BasicType Id "=" BasicValue

BasicType ::= “int” | “bool” | “String”
BasicValue ::= IntValue | BoolValue | StringValue
BoolValue ::= “true” | “false”
ModuleVarDecList ::= “var” ModuleVarDec { ModuleVarDec }
ModuleVarDec ::= Type IdList
Type ::= BasicType
RoutineList ::= { “routine” Id [FormalParamDecList] [ReturnType]
[LocalVarDecList] “is” StatementList “end” Id }
FormalParamDecList ::= “(” FormalParamDec { “,” FormalParamDec } “)”
FormalParamDec ::= Type Id
ReturnType ::= “->” Type
LocalVarDecList ::= “local” LocalDecList { LocalDecList }
LocalDecList ::= Type IdList
StatementList ::= { Statement }
Statement ::= AssignmentStatement | IfStatement | LoopStatement |
RoutineCall | “exiton” |
CompositeStatement | ReturnStatement
AssignmentStatement ::= LeftValue “=” Expression
LeftValue ::= [Id “.”] Id
IfStatement ::= “if” Expression Statement [“else” Statement]
LoopStatement ::= “loop” StatementList “endloop”
RoutineCall ::= Id [“.” Id] [“(” ExpressionList “)”]
CompositeStatement ::= “begin” { Statement } “end”
ExpressionList ::= Expression { “,” Expression }
Expression ::= SimpleExpression [Relation SimpleExpression]
Factor ::= BasicValue | Id [“.” Id] | RoutineCall |
“(” Expression “)” | “not” Factor
HighOperator ::= “∗” | “/” | “and”
LowOperator ::= “+” | “−” | “or”
Relation ::= “==” | “<” | “>” | “<=” | “>=” | “<>”
ReturnStatement ::= “return” Expression
Signal ::= “+” | “−”
SignalFactor ::= [Signal] Factor
SimpleExpression ::= Term { LowOperator Term }
Term ::= SignalFactor { HighOperator SignalFactor }