Small Edison

A Subset of the Edison Programming Language for Arduino Nano

—

SOEN 422 - Fall 2021 Project - Document #1

Dr. Michel de Champlain
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada

November 9, 2021

Abstract

This document presents a subset of the Edison programming language, called Small Edison. The programming language Edison [2, 1, 3] was developed for a multiprocessor system built by Mostek Corporation. This language subset is suitable for teaching embedded systems programming, porting a virtual machine, and loading programs in small-footprint embedded systems, such as the Arduino Nano. Small Edison is block structured and includes modules.

Language Overview

This document presents a subset of the Edison Programming Language, called Small Edison. The overriding concern in the design of Edison was to achieve simplicity in the design of medium-sized programs of 1000-2000 lines of code. This aim led to the selection of a small number of abstract concepts for modular programming.

An Edison program describes operations on named entities called constants, variables, and procedures. The constants and variables have data values of fixed types. The **types** determine the possible values of these entities and restrict the operations that can be performed on them.

The data types **integer**, **boolean**, and **character** are standard types. Other types are introduced by **enum**erating their values or by describing how the values are composed from other values. The composition methods for new types are the well-known **array**s, and **record**s. Every type has a name, and two operands are of the same type only if their types are denoted by the same name and have the same scope.

The computation of new data values from existing values is described by **expressions** in which the operands are constants, variables, and function calls, while the operators are the conventional arithmetic, boolean, and relational operators. The computation of composite values in terms of other values is described by expression lists called constructors.

Statements describe operations such as the assignment of values to variables and the execution of procedures. The **if** and **while** statements describe a choice among substatements to be executed depending on the truth values of expressions.

The named entities and statements of a program are grouped into blocks of two kinds, called procedures and modules. Blocks can be combined to form other blocks in a nested fashion.

A **procedure** is a group of named entities and statements that describe an operation on operands of fixed types. The operands (known as parameters) may vary from one execution of the procedure to another. Each parameter is either a variable or another procedure. The named entities introduced by a procedure

are local to the procedure and cannot be used outside it. Each execution of the procedure creates a fresh instance of its parameters and local variables. These entities disappear again when the execution of the procedure terminates.

A **module** is a group of named entities that fall into two categories: (1) local entities that can only be used within the module, and (2) exported entities that can be used both within the module and within the immediately-surrounding block. The module entities exist as long as the entities of the immediately-surrounding block. A module serves to ensure that its local entities are only operated upon by well-defined procedures which are exported from the module to the immediately-surrounding block. To ensure that the local variables are initialized before they are used, a module includes a statement part that is executed when the module entities are created.

Small Edison Syntax Summary

This section contains the syntax summary of the Small Edison programming language described in a concise fashion using the Extended Backus-Naur Form (EBNF) notation as summarized in Table 1.

Table 1: Notation for Extended Backus-Naur Form

Notation	Meaning
`[ A ]*`	Repetition—zero or more occurrences of `A`
`[ A ]`	Option—zero or one occurrence of `A`
`A B`	Sequence—`A` followed by `B`
`A | B`	Alternative—`A` or `B`
`( A B )`	Grouping—of an `A B` sequence

Compilation Unit

```
Program                    = [ ConstantDeclarationList | TypeDeclaration ]* ProcedureDeclaration .
```

Declarations

```
ConstantDeclarationList    = "const" ConstantDeclaration [ ";" ConstantDeclaration ]* .
ConstantDeclaration        = ConstantName "=" ConstantSymbol .

TypeDeclaration            = EnumerationTypeDeclaration
                           | RecordTypeDeclaration
                           | ArrayTypeDeclaration
                           .

ProcedureDeclaration       = ProcedureHeading [ Declaration ]* StatementPart .
ProcedureHeading           = "proc" ProcedureName [ "(" ParameterList ")" ] [ ":" TypeName ] .

EnumerationTypeDeclaration = "enum" TypeName "(" EnumerationSymbolList ")" .
EnumerationSymbolList      = ConstantName [ "," ConstantName ]* .

RecordTypeDeclaration      = "record" TypeName "(" FieldList ")" .
FieldList                  = FieldGroup [ ";" FieldGroup ]* .
FieldGroup                 = FieldName [ "," FieldName ]* ":" TypeName .
```

```
ArrayTypeDeclaration      = "array" TypeName "[" RangeSymbol "]" "(" TypeName ")" .
RangeSymbol               = ConstantSymbol ":" ConstantSymbol .

Declaration               = ConstantDeclarationList
                          | TypeDeclaration
                          | VariableDeclarationList
                          | ProcedureDeclaration
                          | ModuleDeclaration
                          .

VariableDeclarationList   = "var" VariableGroup [ ";" VariableGroup ]* .
VariableGroup             = VariableName [ "," VariableName ]* ":" TypeName .

ParameterGroup            = [ "var" ] VariableGroup .
ParameterList             = ParameterGroup [ ";" ParameterGroup ]* .

ModuleDeclaration         = "module" [ Declaration | ExportedDeclaration ]* StatementPart .
ExportedDeclaration       = "*" Declaration .
```

Statements

```
StatementPart             = "begin" StatementList "end" .
StatementList             = Statement [ ";" Statement ]* .
Statement                 = VariableSymbol ":=" Expression
                          | ProcedureCall
                          | "if" ConditionalStatementList "end"
                          | "while" ConditionalStatementList "end"
                          | "cobegin" ProcessStatementList "end"
                          | "skip"
                          .

ProcedureCall             = ProcedureName [ "(" ArgumentList ")" ] .
ArgumentList              = Argument [ "," Argument ]* .
Argument                  = Expression | Variable symbol .

ConditionalStatementList  = ConditionalStatement [ "else" ConditionalStatement ]* .
ConditionalStatement      = Expression "do" StatementList .

ProcessStatementList      = ProcessStatement [ "also" ProcessStatement ]* .
ProcessStatement          = ConstantSymbol "do" StatementList .
```

Expressions

```
Expression                = SimpleExpression [ RelationalOperator SimpleExpression ] .
SimpleExpression          = [ "+" | "-" ] Term [ AddingOperator Term ]* .
Term                      = Factor [ MultiplyingOperator Factor ]* .
Factor                    = ConstantSymbol
                          | VariableSymbol
                          | Constructor
                          | Procedure call
                          | "(" Expression ")"
                          | "not" Factor
                          | Factor ":" TypeName
                          .
```

```
ExpressionList              = Expression [ "," Expression ]* .
Constructor                 = TypeName [ "(" ExpressionList ")" ] .
RelationalOperator          = "=" | "<>" | "<" | "<=" | ">" | ">=" .
AddingOperator              = "+" | "-" | "or" .
MultiplyingOperator         = "*" | "div" | "mod" | "and" .
```

Symbols

```
VariableSymbol              = VariableName
                            | "val" ProcedureName
                            | VariableSymbol "." FieldName
                            | VariableSymbol "[" Expression "]"
                            | VariableSymbol ":" TypeName
                            .

ConstantSymbol              = ConstantName | Numeral | CharacterSymbol .
CharacterSymbol             = "'" AsciiPrintableCharacter "'" | "char" "(" Numeral ")" .
Numeral                     = Digit [ Digit ]* .
Name                        = Letter [ Letter | Digit | "_" ]* .
Digit                       = "0" .. "9" .
Letter                      = "a" .. "z" | "A" .. "Z" .
```

References

[1] Per Brinch Hansen. The Design of Edison. *Software, Practice and Experience*, 11(4):363–396, April 1981.

[2] Per Brinch Hansen. Edison — a Multiprocessor Language. *Software, Practice and Experience*, 11(4):325–361, April 1981.

[3] Per Brinch Hansen. Edison Programs. *Software, Practice and Experience*, 11(4):397–414, April 1981.