The C#Minor Programming Language

P.D. Terry, Computer Science Department, Rhodes University

27 June 2003

1 Introduction

C#Minor is a small programming language that has evolved from C# and Java. It has the appearance of being a small subset of either of those languages. While it supports the definition of simple classes and one-dimensional array types and the creation and manipulation of objects of those types, there is no provision for inheritance, polymorphism or overloading, for handling exceptions or multidimensional arrays, and no support for the C# concepts of delegates, indexers, enumerations, properties and events.

C#Minor was developed as a possible vehicle for teaching compiler construction using the compiler generators Coco/R for C# and for Java (Mössenböck, 2004, Terry, 2005). Coco/R creates recursive descent compilers from attributed LL(1) grammars. Various implementations of C#Minor are available, either targeting the Java Virtual Machine (JVM) by producing JVM assembly code for further processing with the Oolong assembler (Engel, 1999), or targeting Microsoft's .NET Common Language Runtime (CLR) by producing CIL assembly code for assembly with Microsoft's ILASM assembler.

Due to the limitations of LL(1) parsing, various features of C# and Java have been compromised in the design of C#Minor. Nevertheless, valid C#Minor programs should also be valid C# programs and, with a few changes in keywords, should also be valid Java programs.

This specification of C#Minor is modelled on the Modula-2 and Oberon specifications (Wirth (1985), Reiser and Wirth (1992)), but is not intended as a programmer's tutorial. It is intentionally kept concise. Its function is to serve as a reference for programmers, implementors, and manual writers. What remains unsaid is mostly left so intentionally, either because it is derivable from stated rules of the language, or because it would require a general commitment in the definition when a general commitment appears unwise.

2 Syntax

A language is an infinite set of sentences, namely the sentences well formed according to its syntax. Each sentence is a finite sequence of symbols from a finite vocabulary. The vocabulary of C#Minor consists of identifiers, numbers, strings, operators, delimiters, and comments. These are called lexical symbols and are composed of sequences of characters. (Note the distinction between symbols and characters.)

To describe the syntax the variant of extended Backus-Naur formalism called Cocol/R is used. This is described in full detail elsewhere (Terry, 2005). Brackets [and] denote optionality of the enclosed sentential form, and braces { and } denote its repetition (possibly 0 times). Syntactic entities (non-terminal symbols) are denoted by English words expressing their intuitive meaning. Symbols of the language vocabulary (terminal symbols) are denoted by strings enclosed in quote marks (these include words written mainly in lower-case letters, so-called reserved keywords).

3 Vocabulary and representation

The representation of symbols in terms of characters is defined using the ASCII set. Symbols are identifiers, numbers, string literals, character literals, operators, delimiters, and comments.

The following lexical rules must be observed. Blanks and line breaks may appear between symbols but must not occur within symbols (except that line breaks are allowed in comments, and blanks are allowed within string and character literals). They are ignored unless they are essential for separating two consecutive symbols. Capital and lower-case letters are considered as being distinct.

Comments may be inserted between any two symbols in a program. They are arbitrary character sequences either opened by the bracket `/*` and closed by `*/`, or opened by the bracket `//` and closed at the end of that line. Comments do not affect the meaning of a program.

```
CHARACTERS
  cr          = CHR(13) .
  lf          = CHR(10) .
  backslash   = CHR(92) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  stringCh    = ANY - '"' - control - backslash .
  charCh      = ANY - "'" - control - backslash .
  printable   = ANY - control .

TOKENS
  identifier = letter { letter | digit | "_" } .
  number     = digit { digit } .
  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" charCh | backslash printable "'" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)
```

Identifiers are sequences of letters, underscores and digits. The first character must be a letter.

Examples:

```
x   scan   CSharpMinor   Get_Symbol   firstLetter
```

Numbers are (unsigned) integers. Integers are sequences of digits, and have their usual decimal interpretation. (The implementation uses 32-bit arithmetic.)

String literals are sequences of zero or more graphic characters or escape sequences enclosed in quote marks ("). The number of characters in a string is called the length of the string. Strings can be used only in fairly limited contexts. A string literal may not extend over a line break in the source text.

Character literals are denoted by a single graphic character or a single escape sequence between single quote marks ('). Character literals denote the integer value of the corresponding ASCII character.

Within a string or character literal the following escape sequences denote non-graphical characters

```
\0         NUL character    (CHR(0))
\b         Backspace        (CHR(8))
\t         Horizontal tab   (CHR(9))
\n         Line feed        (CHR(10))
\f         Form feed        (CHR(12))
\r         Carriage return  (CHR(13))
\"         Quotation mark   (CHR(34))
\'         Apostrophe       (CHR(39))
\x         (where x is not 0, b, t, n, f or r) denotes x itself
```

(Within a string `\n` denotes the sequence Carriage return, Line feed, on Microsoft systems.)

Examples:

```
"C#Minor will become major"       "He said\"food!\" and fed a line\n"
```

Operators and delimiters are the special characters, character pairs, or reserved keywords listed below. The reserved words cannot be used in the role of identifiers. Those in parentheses are not currently keywords, but are reserved for use in later extensions .

```
!     <     (     IO.ReadBool     char         private       (case)
&&    <=    )     IO.ReadChar     class        public        (catch)
||    ==    {     IO.ReadInt      const        return        (continue)
+     >     }     IO.ReadLine     do           static        (enum)
-     >=    []    IO.ReadString   else         string        (finally)
*     =     [     IO.ReadWord     false        this          (for)
/     .     ]     IO.Write        final        true          (goto)
%     ,           String          if           void          (out)
++    :           bool            int          while         (ref)
--    ;           boolean         new                        (switch)
                  break           null                       (throw)
                                                             (try)
```

Some of the keywords are synonyms - for example `bool` and `boolean`, `string` and `String`, `const` and

`final`.

The keywords of the form `IO.xxx` are introduced to allow the implementation easily to produce code suitable for linking to a separately compiled library module (see section 17).

4 Declarations and scope rules

Every identifier occurring in a program must be introduced by a declaration. Declarations also serve to specify certain permanent properties of the entity denoted by that identifier, such as its type (see section 7), and whether it is a constant, a variable, a parameter, a class, an object, a field, a method, or an array.

The identifier is then used to refer to the associated entity. This is possible only in those parts of a program that are within the scope of the declaration. No identifier may denote more than one entity within a given scope. The scope extends textually from the point of the declaration to the end of the class, method or block in which the declaration has been made and to which the entity is said to be local. This rule is augmented in specific ways that are clarified below.

5 Programs

A program is defined by a set of class declarations.

```
Program = { ClassDeclaration } EOF .
```

6 Class declarations

A class declaration defines a new data type and, in particular, a set of data members (fields) and a set of function members (methods) that define the operations permitted on the data members.

```
ClassDeclaration  = Modifiers "class" ClassIdentifier "{"
                    { ConstDeclarations | FieldDeclarations | MethodDeclaration | ClassDeclaration }
                    "}" .
ConstDeclarations = Modifiers ( "final" | "const" ) BasicType OneConst { "," OneConst } ";" .
OneConst          = ConstIdentifier "=" Expression .
FieldDeclarations = Modifiers Type OneField { "," OneField } ";" .
OneField          = FieldIdentifier [ "=" Expression ] .
MethodDeclaration = Modifiers Type MethodIdentifier "(" ParamList ")"  Block .
Modifiers         = [ "public" | "private" ] [ "static" ] .
ClassIdentifier   = identifier .
ConstIdentifier   = identifier .
FieldIdentifier   = identifier .
MethodIdentifier  = identifier .
```

Once a class has been declared, fields, variables and parameters of that type may be declared, and assigned the values of objects created by instantiation of the class (see section 13.5).

Access to the members of a class may be controlled by the use of a set of *Modifiers* defined on each member. If a member is marked as `private` (or left unmarked) it may be accessed only within that class. If a member is marked as `public` it is accessible within the declaring class and, by appropriate selection, from within other classes. Unless marked as `static`, members are known as *instance members* and may only be accessed through appropriate selection by instantiated objects of that class; if a member is marked as `static` it is known as a *class method* or *class field*, and is associated with the class as a whole. Member selection is achieved through *Designators* (see section 14).

No member may have an identifer that is the same as the *ClassIdentifier* of its enclosing class. The scope of member identifiers extends into later methods of the class, save that such methods are free to redeclare these identifiers to stand for formal parameters, whose scope extends to the end of those methods.

7 Type

In general, a type determines the set of values which fields, variables and parameters declared to be of that type

may assume, and the operators that are applicable to such fields, variables and parameters.

```
BasicType  = "int" | "char" | "bool" | "boolean" | "void" | "string" | "String" .
SimpleType =  BasicType | ClassIdentifier .
Type       =  SimpleType [ "[]" ] .
```

The `void` type has a special meaning in the context of method declarations and is not used in other contexts.

C#Minor distinguishes between *value types* and *reference types*, and supports several fundamental value types. The type denoted by `int` represents the set of 32-bit integer values {-2147483648 … 2147483647}. The type denoted by `bool` or `boolean` represents the set of Boolean values {`false, true`}. The type denoted by `char` denotes the set of ASCII characters. The type denoted by `string` or `String` denotes the set of character literals of arbitrary length and is of limited application in this release.

A type denoted by a *ClassIdentifier* is said to be a *reference type*. Values of such types (objects) are constructed as components of expressions by instantiation, using the keyword `new` (see section 13.5).

A type that incorporates the `[]` token is said to be an *array reference type*. Arrays are constructed as components of expressions by using the keyword `new` (see section 13.5).

Two entities are said to be *type-compatible* (for the purposes of assignment, comparison or when passed as parameters) if they are of the same type. In general the type of a value defined by an *Expression* (see section 13) is computed from its operators and the types of its operands; the type of a variable, parameter or field is determined from the type used in its declaration. However, the value denoted by `null` is a value of all reference types, and the `int` and `char` types are compatible in that values of one may be compared with values of the other, and values of the `char` type may be assigned to variables, parameters and fields of the `int` type.

8 Constant declarations

A constant declaration permanently associates an identifier with a constant value, and may appear within a *ClassDeclaration*, or within the *Block* of a *MethodDeclaration*.

```
ConstDeclarations = Modifiers ( "final" | "const" ) BasicType OneConst { "," OneConst } ";" .
OneConst          = ConstIdentifier "=" Expression .
Modifiers         = [ "public" | "private" ] [ "static" ] .
BasicType         = "int" | "char" | "bool" | "boolean" | "void" | "string" | "String" .
ConstIdentifier   = identifier .
```

No modifiers are permitted when the declarations are within the *Block* of a *MethodDeclaration*, and the `static` modifier is only permitted in conjunction with the keyword `final` (for compatability with Java code) in declaring constants within *ClassDeclarations*. The type of the *Expression* must be compatible with the *BasicType* used in the declaration, and the value of the *Expression* must be computable at compile-time (that is, its operands must all be constants).

Examples:

```
const int MAX = 100, MAXPLUS1 = MAX + 1;
const bool YES = true;
const char CapitalA = 'A';
const string FavouriteLanguage = "C#Minor";
```

Note: C# appears to allow the declaration of constants of reference types. Apart from strings the only *Expression* permitted appears to be denoted by `null`, and so virtually any attempt to reference such quantities thereafter raises exceptions. Hence it has not been seen necessary to allow this flexibility in C#Minor.

9 Field declarations

Fields are data members of a class whose values may be changed by execution of the program. Field declarations serve to introduce fields and to associate them with identifiers that must be unique within the given scope. They also serve to associate a fixed data type with each field so introduced.

```
FieldDeclarations = Modifiers Type OneField { "," OneField } ";" .
OneField          = FieldIdentifier [ "=" Expression ] .
Modifiers         = [ "public" | "private" ] [ "static" ] .
FieldIdentifier   = identifier .
```

Fields whose identifiers appear in the same list are all of the same type.

When the enclosing class is loaded, class fields (those marked `static`) are deemed to have initial values represented by zero or `null`, except for those fields that have been assigned the values of expressions within the field declaration sequence where they were declared. Similarly, when a class is instantiated (see section 13.5) instance fields (those not marked `static`) are deemed to have initial values represented by zero or `null`, except for those fields that have been assigned the values of expressions within the field declaration sequence where they were declared.

Examples:

```
class Sieve {
  const SIZE = 100;
  public bool Started = true;
  bool[] sieve = new bool[SIZE];
}
```

10 Method declarations

A method declaration serves to define operations that may be performed on fields of the enclosing class, on `public` accessible fields of other classes or objects, or on values passed to the method as arguments.

```
MethodDeclaration = Modifiers Type MethodIdentifier "(" ParamList ")"  Block .
Modifiers         = [ "public" | "private" ] [ "static" ] .
MethodIdentifier  = identifier .
ParamList         = [ OneParam { "," OneParam } ] .
OneParam          = Type ParamIdentifier .
ParamIdentifier   = identifier .
Block             = "{" StatementSequence "}" .
```

A restriction is imposed in the current implementation which requires each identifier in a program to be "declared" before it is "used". In many cases this is easily achieved by constituting a program as a set of *ClassDeclarations* and *MethodDeclarations*, textually declared in an order that meets this requirement. In some situations - typically those in which mutually recursive methods need to invoke one another - this will be found to be impossible.

A `void` method is activated by a method call that is a form of *Statement* (see section 15.3). A method of any other type is activated by a method call that forms a constituent part of an *Expression* (see section 13), and yields a result that is an operand of that expression.

The *Block* uniquely associated with each method incorporates a collection of declarations of constants and local variables (whose values are said to constitute the program state), and a sequence of other statements purpose is to alter the program state by manipulating the local variables, formal parameters and those fields of the set of methods to which that method has access.

The *ParamList* associated with a method provides one mechanism by which data may be transmitted from one method to another (see sections 13.3 and 15.3).

One method is uniquely designated as the `Main` method. This is the first to be executed when the program as a whole is executed. It must be of type `void` and often has no formal parameters; that is, it is declared with the header `public static void Main()`. Another header, which allows the program access to the command line parameters used when execution begins, is `public static void Main(string[] args)`.

Note: in C#Minor the names `main` and `Main` are indistinguishable, for compatibility with both C# and Java.

11 Formal Parameters

Formal parameters that are a component part of a *MethodDeclaration* are specified in a (possibly empty) list of

identifiers that denote actual parameters that are specified only when a method is called. The correspondence between formal and actual parameters is established only when the call takes place.

```
ParamList       = [ OneParam { "," OneParam } ] .
OneParam        = Type ParamIdentifier .
ParamIdentifier = identifier .
```

The type of a formal parameter is specified in its declaration; this type must be compatible with the type of the actual parameter used in the *MethodCall*. There must be as many arguments in the *ArgList* of the *MethodCall* as there are parameters in the *ParamList* of the *MethodDeclaration*. In particular, a method declared without parameters has an empty parameter list, and must be invoked by a *MethodCall* whose actual *ArgList* is empty.

Examples:

```
static int larger(int a, int b) {
// Returns larger of the two arguments
  if (a > b) return a; else return b;
}

static int sum(int[] list, int n) {
// Returns sum of n elements - list[0] ... list[n-1]
  int total = 0;
  int i = 0;
  while (i < n) { total = total + list[i]; i++; }
  return total;
}
```

12 Variable declarations

Variables are those data items declared local to a method or statement *Block* whose values may be changed by execution of the program. Variable declarations serve to introduce variables and to associate them with identifiers that must be unique within the given scope. They also serve to associate a fixed data type with each variable so introduced.

```
VarDeclarations    = Type OneVar { "," OneVar } ";" .
OneVar             = VariableIdentifier [ "=" Expression ] .
VariableIdentifier = identifier
```

The scope of a variable declaration extends from the point of declaration to the end of the *Block* in which the declaration appears. No variable may have the same name as any other *ClassIdentifier, Metodidentifier, VariableIdentifier, ParamIdentifier* or *ConstIdentifier* already visible at the point of declaration.

Variables whose identifiers appear in the same list are all of the same type.

When the statement sequence that is defined by a *Block* is activated, all variables are deemed to have initially undefined values, except for those scalar variables that are assigned the values of expressions within the variable declaration sequence where they were declared.

Examples:

```
void method() {
  int i, j = 8;
  boolean Started = true, Suitable;
  char Letter, reply;
  int[] list = null;
  Node n = new Node();
}
```

13 Expressions

Expressions are constructs denoting rules of computation whereby the current values of variables, fields, parameters, constants and values returned by method calls may be combined to derive other values by the application of operators. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands where the normal rules of precedence are unsuitable on their own.

```
Expression     = AndExp { "||" AndExp } .
AndExp         = EqlExp { "&&" EqlExp } .
EqlExp         = RelExp { EqlOp RelExp } .
RelExp         = AddExp [ RelOp AddExp ] .
AddExp         = MulExp { AddOp MulExp } .
MulExp         = Factor { MulOp Factor } .
Factor         = Primary | "+" Factor | "-" Factor | "!" Factor .
Primary        =   Designator
                 | MethodCall
                 | TypeCast
                 | InputValue
                 | ArrayValue
                 | ObjectValue
                 | "true" | "false" | "null" | charLit | number | stringLit
                 | "(" Expression ")" .
Designator     = ( identifier | "this" ) [ Selector ] .
Selector       = ( "." identifier | "[" Expression "]" ) [ Selector ] .
MethodCall     = Designator "(" ArgList ")" .
ArgList        = [ OneArg { "," OneArg } ] .
OneArg         = Expression .
SimpleType     = BasicType | ClassIdentifier .
TypeCast       = "(" "char" ")" Factor | "(" "int" ")" Factor .
InputValue     = (   "IO.ReadInt" | "IO.ReadBool" | "IO.ReadChar"
                   | "IO.ReadString" | "IO.ReadLine" | "IO.ReadWord" ) "(" ")"
ArrayValue     = "new" SimpleType "[" Expression "]" .
ObjectValue    = "new" ClassIdentifier "(" ")" .
EqlOp          = "==" | "!=" .
RelOp          = "<" | "<=" | ">" | ">=" .
AddOp          = "+" | "-" .
MulOp          = "*" | "/" | "%" .
```

13.1 Operands

With the exception of literals, many operands in expressions are denoted by designators (see section 14).

If `A` designates an array, then `A[E]` denotes that element of `A` whose index is the current value of the expression `E`. `E` must be of integer or character type, and the value of `E` must lie within the range of possible values for the index of `A`. This range is from `0 ... ArraySize - 1`, as specified when `A` was instantiated.

If the designated entity is a field, parameter, variable or an array element, then, when used as an operand, the designator refers to that entity's value, and the type of the operand is the type of that entity. An operand that is a literal constant denotes a value that is the value of that constant, and the type of the operand is the type of that literal constant.

Operands may also be designated by method calls (see section 13.3) or by the values of new instantiations of class and array reference types (see section 13.5).

Note: in C# and Java it is illegal for an expression to incorporate operands whose values are undefined at the point where the expression is to be evaluated. No such check is currently carried out in the implementations of C#Minor.

13.2 Operators

The syntax of expressions distinguishes between several classes of operators with different precedences (binding strengths). The syntax and operator precedence are modelled on the baroque C++/Java/C# model. The unary operators have the highest precedence, followed by multiplication operators, addition operators, relational operators, and logical operators. Operators of the same precedence associate from left to right. Thus, for example, `x - y - z * w` stands for `(x - y) - (z * w)`.

The available operators are listed in the following tables.

13.2.1 Logical operators

symbol	result
`||`	logical disjunction (Boolean OR)
`&&`	logical conjunction (Boolean AND)
`!`	negation

These operators apply only when their operands are of the Boolean type, and yield a Boolean result.

`p || q`	stands for "if `p` then `true`, else `q`"
`p && q`	stands for "if `p` then `q`, else `false`"
`! p`	stands for "not `p`"

13.2.2 Arithmetic operators

symbol	result
`+`	sum
`-`	difference
`*`	product
`/`	quotient
`%`	modulus

These operators apply only to operands of integer or character type, and yield a result of integer type. When used as operators with a single operand `+` denotes the identity operation and `-` denotes sign inversion.

The operator `/` produces a result that is truncated towards zero. The operator `%` produces a result so that the following relation holds for any dividend `x` and divisor `y`:

```
x  =  (x / y) * y  +  (x % y)
```

Examples

```
 x       y     x / y   x % y

 12      7       1       5
 12     -7      -1       5
-12      7      -1      -5
-12     -7       1      -5
```

13.2.3 Relational operators

symbol	relation
`==`	equal
`!=`	unequal
`<`	less
`<=`	less or equal
`>`	greater
`>=`	greater or equal

These operators yield a result of Boolean type. The ordering operators `<`, `<=`, `>` and `>=` apply to operands of the integer and character types. The relational operators `==` and `!=` also apply when both operands are of the Boolean type or of the same reference (class or array) type (the comparison is between the values of the references, and not between the individual elements or fields of the objects referred to). The operators are not defined on values of `string` type. The operands of a relational operator may be evaluated in any convenient order.

Examples of expressions:

```
1996                    (Integer)
i / 3                   (Integer)
!Started || Suitable    (Boolean)
(i + j) * (i - j)       (Integer)
0 <= i && i < max       (Boolean)
```

13.3 Method calls

A *MethodCall* that serves to activate a non-`void` method may appear as an operand in an *Expression*, where it represents the value computed and returned by that particular activation of the designated method.

```
MethodCall = Designator "(" ArgList ")" .
ArgList    = [ OneArg { "," OneArg } ] .
OneArg     = Expression .
```

The call may contain an *ArgList* of arguments or actual parameters which are substituted in place of the corresponding *ParamList* defined in the *MethodDeclaration*. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. The types of corresponding actual and formal parameters must be the same, and there must be as many actual arguments in the *ArgList* as there are formal parameters in the *ParamList* of the method called.

Note: C#Minor does not support method overloading, whereby two or more methods may share a common name and return type but differ in the number of arguments that may be supplied.

There are two kinds of actual parameters. If the formal parameter is of a reference type, the actual parameter, while syntactically an *Expression*, must be a *Designator* denoting an object of that type (or a formal parameter of that type in the method in which the *MethodCall* is an operand). In the case of a value parameter, the actual parameter may be a more general *Expression*. Each expression is evaluated prior to the method activation, and the resulting value is assigned to the formal parameter, which then constitutes a local variable of the called method.

The use of a method identifier in an *Expression* within the *Body* associated with the definition of that method implies a recursive activation of that method.

13.4 Predefined input methods

An operand in an expression may consist of a method call to an input routine provided in an external library.

```
InputValue = (   "IO.ReadInt"    | "IO.ReadBool" | "IO.ReadChar"
               | "IO.ReadString" | "IO.ReadLine" | "IO.ReadWord" ) "(" ")"
```

The value of such an operand is obtained when the runtime system scans the standard input stream and converts a textual representation of a value into the value itself.

The implementation requires that a library module `IO.dll` or `IO.class` be available for linking with the code produced by the C#Minor compiler (see section 17).

13.5 Object instantiation

Operands in expressions may have values that are obtained when a new object of a reference or array reference type is created.

```
ArrayValue = "new" SimpleType "[" Expression "]" .
```

An array value allows for the creation of a new object of an array reference type, whose elements are all of the type specified by *SimpleType*. The number of elements is specified by the value of the *Expression*, which must yield a positive value of integer type. The value of the operand is the reference to the newly created object. Typically this is assigned to a field or variable declared to be of a type specified as `SimpleType[]`, whereafter the elements of the array object may be accessed by use of appropriate *Designators* (see section 14). The number of elements in the array object is known as its length, and may be retrieved by a construction of the form `ArrayName.Length`.

```
ObjectValue = "new" ClassIdentifier "(" ")" .
```

An object value arises from the creation of a new instance of the class specified by *ClassIdentifier*. Typically this is assigned to a field or variable declared to be of that type, whereafter the instance fields and methods of the object may be accessed by use of appropriate *Designators* (see section 14).

Examples:

```
int[] IntList = new int[100];
Class c = new Class();
Class[] cList = new Class[2*n];
int i = 0;
while (i < cList.Length) { cList[i] = new Class(); i++; }
```

Note: The class value constructors allowed in C#Minor programs cannot have parameters. However, simple classes can make available (`static`) class methods that perform such construction for clients. This, coupled with the fact that when an object is created its fields may be initialized to values specified in the *FieldDeclarations*, handles most cases of simple object creation.

Example:

```
class One {
  int x = 0;

  public static One MakeOne(int x) {
    One temp = new One();
    temp.x = x;
    return temp;
  }

}
```

13.6 Type-casting

Type-casting allows for the explicit conversion of an operand of one type to the equivalent value of another type.

```
TypeCast = "(" "char" ")" Factor | "(" "int" ")" Factor .
```

C#Minor provides only for type-casting from an integer value to a character value or from a character value to an integer value. Type-casting is dependent on the value being cast being within range of the target type. This will always be true for an integer cast, but a character cast may result in loss of precision.

Examples:

```
char ch = 'a'; IO.Write( (int) ch );  // outputs the value 97
int i = 65;    IO.Write( (char) i );  // outputs the character A
```

Note: it is not possible to cast between Boolean values and other types.

14 Designators

Designators provide the mechanism for specifying the methods, simple variables, constants and parameters needed in assignment statements, expressions and method calls. They also provide mechanisms for selecting elements from objects of array reference types, instance fields and methods from objects, and static fields and methods from classes.

```
Designator = ( identifier | "this" ) [ Selector ] .
Selector   = ( "." identifier | "[" Expression "]" ) [ Selector ] .
```

The syntax for *Designator* allows for complex sequences incorporating many identifiers. However these are subject to the following complicated constraints:

In the absence of any *Selector* component, the single identifier may correspond only to a *MethodIdentifier, VariableIdentifier, ParamIdentifier* or *ConstIdentifier* (but not a *ClassIdentifier*). Within an instance method of a class it may also correspond to an instance *FieldIdentifier*, and within a (`static`) class method it may also

correspond to a (`static`) class *FieldIdentifier*. When such a designator appears as an operand in an *Expression*, it stands for the value returned by a call of that method, or the value of the variable, parameter, constant or field respectively. In the case where it is a *VariableIdentifier, ParamIdentifier* or *FieldIdentifier*, the designator may also appear as the target of an *Assignment*, when it stands for the address (l-value) associated with that identifier. The keyword `this` denotes a direct reference to a class instance and may only be used within an instance method of that class.

If any *Selector* components containing further identifiers are present, the leading identifier in the whole *Designator* may correspond only to a *ClassIdentifier, FieldIdentifier, VariableIdentifier* or *ParamIdentifier*. The final identifier in the *Selector* component must then correspond to a *ConstIdentifier, MethodIdentifier* or *FieldIdentifier*, and any intermediate identifiers must correspond to *FieldIdentifiers*. If the leading identifier is a *ClassIdentifier*, this final identifier must designate a *ConstIdentifier* or static *MethodIdentifier* or *FieldIdentifier* associated with that class. If the leading component is denoted by the keyword `this`, or by a *VariableIdentifier* or *ParamIdentifier*, the last identifier in the *Selector* component must designate a *ConstIdentifier* or instance *MethodIdentifier* or *FieldIdentifier* associated with the object designated by the preceding components of the *Designator*.

A *Selector* component incorporating a bracketed *Expression* designates an array element of the variable or field designated by the preceding components of the designator. In this case the identifier immediately preceding the bracketed *Expression* must correspond to a *FieldIdentifier, VariableIdentifier* or *ParamIdentifier* of an appropriate array reference type.

When a multicomponent designator appears as an operand in an *Expression*, it stands for a value returned by a call of any method so designated, or the value of any field, variable, parameter, array element or constant so designated. The last identifier to appear in a multicomponent designator is of special significance. In the case where the last identifier is a *FieldIdentifier*, *VariableIdentifier* or *ParamIdentifier* the designator may also appear as the target of an *Assignment*, when it stands for the address (l-value) associated with the designated entity.

Examples:

```
class P {
  public int[] p = new int[10];
}

class Q {
  public int x;
  static int y;
  int[] list;
  P q;

  void InstanceMethod(int[] a, int j) {
    this.x = x + a[j + 4];
  }

  static void StaticMethod(Q q) {
    y = q.list[q.x];
  }

  public static void Main() {
    Q[] QList = new Q[10];
    QList[1] = new Q();
    Q.StaticMethod(QList[1]);
    QList[1].InstanceMethod(new int[7], QList[1].q.p[3]);
  }
}
```

15 Statements

Statements denote actions or the declaration and possible initialization of local variables and constants. Apart from declarative statements, a distinction is drawn between elementary and structured action statements.

Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the void method call, the write statement, the return statement, the break statement and the empty statement.

Structured statements are composed of parts that incorporate further statements. They are used to express sequencing, as well as conditional, selective, and repetitive execution.

```
Statement      =   Block | Assignment | VoidMethodCall | WriteStatement
                 | IfStatement | WhileStatement | DoWhileStatement
                 | BreakStatement | ReturnStatement
                 | ConstDeclarations | VarDeclarations | ";" .
VoidMethodCall = MethodCall .
```

15.1 Blocks - statement sequences

```
Block             = "{" StatementSequence "}" .
StatementSequence = { Statement } .
```

A *Block* statement may be used to group several statements and declarations into one indivisible unit. Block statements are particular useful as components of structured statements.

Statement sequences within a *Block* statement denote the sequence of actions specified by the component statements, executed in the order specified by the sequence.

Note that a *Block* may incorporate the declarations of further constants and variables whose scope is restricted to that block, and to blocks introduced within that block, subject to the rule that the redeclaration of an identifier in an inner block is, as in Java, not permitted (as it would be in some other languages).

Note that although blocks can be nested, methods cannot be nested.

15.2 Assignments

An *AssignmentStatement* denotes the replacement of the value of a variable with a new value, and results in a change to the state of a program.

```
AssignmentStatement = Variable ( "=" Expression | "++" | "--" ) ";" .
Variable            = Designator .
```

If the *Expression* is present, the assignment serves to replace the current value of the designated *Variable* by the value specified by the *Expression*. The simple assignment operator is written as "=" and pronounced as "becomes". The types of the *Expression* and of the *Variable* must be the same, save that the value of a character expression may be directly assigned to an integer designator, and the value null may be assigned to a designator of any reference type.

Statements of the form `Variable++;` and `Variable--;` are semantically equivalent to the statements `Variable = Variable + 1;` and `Variable = Variable - 1;` respectively. In this case the `Variable` must be of the integer or character type.

Examples:

```
i = 0;
suitable = i == j;
created = C.node != null;
l = j + i + k % 3;
list[i]++;
bankBalance = bankBalance - car.price;
```

In C#Minor, *Expressions* and *Statements* are clearly distinguishable, which is not the case in C++, C# and Java, where "assignment expressions" (with side effects) are, nevertheless, usually taught to the neophyte to be "assignment statements". This means that statements (expressions) of the form `a = b = c` are not permitted. Furthermore, the use of the `++` and `--` operators has been restricted in a way that renders what in some languages are regarded as expressions of the form `i++` syntactically to become "assignment statements".

15.3 Void method calls

A *VoidMethodCall* serves to activate a `void` method.

```
VoidMethodCall   = MethodDesignator "(" ArgList ")" .
MethodDesignator = Designator .
ArgList          = [ OneArg { "," OneArg } ] .
OneArg           = Expression .
```

The designated method must have been declared of type `void`. There is no concept, as in C# or Java, of invoking a non-void method and silently discarding the return value.

The call may contain an *ArgList* of arguments or actual parameters which are substituted in place of the corresponding *ParamList* defined in the *MethodDeclaration*. The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. The types of corresponding actual and formal parameters must be the same, and there must be as many actual arguments in the *ArgList* as there are formal parameters in the *ParamList* of the method called.

Note: C#Minor does not support method overloading, whereby two or more `void` methods may share a common name but differ in the number of arguments that may be supplied.

There are two kinds of actual parameters. If the formal parameter is of a reference type, the actual parameter, while syntactically an *Expression*, must be a *Designator* denoting an object of that type (or a formal parameter of that type in the method in which the *VoidMethodCall* is a statement). In the case of a value parameter, the actual parameter may be a more general *Expression*. Each expression is evaluated prior to the method activation, and the resulting value is assigned to the formal parameter, which then constitutes a local variable of the called method.

The use of a method identifier in a *VoidMethodCall* within the *Body* associated with the definition of that `void` method implies a recursive activation of that method.

15.4 If statements

An *IfStatement* specifies the conditional execution of a guarded *Statement*.

```
IfStatement = "if" "(" Condition ")" Statement [ "else" Statement ] .
Condition   = Expression .
```

The *Expression* (guard) forming the *Condition* must yield a Boolean result. If the guard evaluates to `true` the guarded *Statement* following the *Condition* is executed. If the guard evaluates to `false`, the *Statement* following the symbol `else` is executed, if there is one.

Note: A statement of the form `if (p) if (q) a; else b;` displays the "dangling else" ambiguity, which is resolved by treating the statement as though it were rewritten `if (p) { if (q) a; else b; }` and not as though it were rewritten `if (p) { if (q) a; } else b;` That is, an unbalanced `else` clause is bound to the most immediately preceding `if`.

Examples:

```
if (i > max) {
  IO.Write("limit exceeded)"; return;
}
if (j % 3 == 0) IO.Write("divisible by 3");
else IO.Write("not divisible by 3");
```

15.5 While statements

A *WhileStatement* is one form of specifying the repetition of an associated statement.

```
WhileStatement = "while" "(" Condition ")" Statement .
Condition      = Expression .
```

The *Expression* that is the *Condition* must yield a Boolean result. If this expression yields `true`, the associated *Statement* is executed. Evaluation of the condition is performed once at the start of each complete iteration, and the test and statement execution are repeated as long as the *Condition* yields `true`.

Example:

```
while (j > 0) {
  j = j / 2; i = i + k;
}
```

15.6 Do-While Statements

A *DoWhileStatement* is one form of specifying the repetition of an associated statement.

```
DoWhileStatement =  "do"  Statement  "while" "(" Condition ")" ";" .
Condition        =  Expression .
```

The *Expression* must yield a Boolean result. A *DoWhileStatement* specifies the repeated execution of the associated *Statement* as long as the *Condition* is satisfied. Evaluation of the condition is performed once at the end of each complete iteration and so the *Statement* is executed at least once.

Example:

```
do {
  j = j / 2; i = i + k;
} while (j > 0);
```

15.7 Write statements

A *WriteStatement* specifies an expression whose value is to be computed and then represented textually on the standard output sink external to the program.

```
WriteStatement = "IO.Write" "(" Expression [ "," MinimumWidth ] ")" ";" .
MinimumWidth   = Expression .
```

The *Expression* must be of type `int`, `char`, `bool` or `string`. The optional second *MinimumWidth* must be of integer type and specifies the minimum width of field to be used for the output, save that the special value of 0 means "precede with exactly one space". By default a minimum width of 0 is used for integer and Boolean results, and a width of 1 for and string and character results. Positive values of *MinimumWidth* specify that right justification is to be used; negative values specify that left justification is to be used.

The implementation requires that a library module `IO.dll` or `IO.class` be available for linking with the code produced by the C#Minor compiler (see section 17).

Example:

```
IO.Write("The result is "); IO.Write(i + j);
IO.Write(" The answer is "); IO.Write(i % k, 4); IO.Write("\n");
```

15.8 Return statements

A *ReturnStatement* causes immediate termination of execution of the method in which it appears, returning control to the invoking environment.

```
ReturnStatement = "return" [ Expression ] ";" .
```

In the case of `void` methods the *Expression* must be absent. In all other methods it must be present, and defines the value to be returned as the result of invoking that method. The type of this *Expression* must be compatible with that of the method as specified in the *MethodDeclaration*. A method may incorporate several *ReturnStatements*, although, of course, only one can be executed in any particular activation of the method. In `void` methods an implicit *ReturnStatement* occurs at the end of the method *Block*.

15.9 Break Statements

A *BreakStatement* may be used only within the associated *Statement* (more generally a *StatementSequence*) of a loop construction (either a *WhileStatement* or *DoWhileStatement*) and causes immediate termination of execution of the loop and continuation with the statement following the loop.

```
BreakStatement = "break" ";" .
```

Example:

```
int total = 0, i;
while (true) { // indefinite loop
  i = IO.ReadInt();
  if (i == terminator) break;
  total = total + i;
}
IO.Write("total = "); IO.Write(total);
```

15.10 Empty statements

An *EmptyStatement* causes no change to the program state.

```
EmptyStatement =  ";" .
```

The *EmptyStatement* is included in order to relax punctuation rules in statement sequences.

15.11 Missing statements

There are no `for` or `continue` statements in this release of the language.

There is no `switch` statement in this release of the language.

16 Complete examples

The first example shows the implementation and use of a three-dimensional vector class, and illustrates the way in which an object may be created and its various fields initialized, as well as the way in which an object may be cloned or the intrinsic values of two objects compared.

```
class Vector {
// Simple 3d vector class
  int x, y, z;

  public static Vector NewVector(int x, int y, int z) {
    Vector v = new Vector();
    v.x = x; v.y = y; v.z = z;
    return v;
  }

  public static Vector ReadVector() {
    Vector v = new Vector();
    v.x = IO.ReadInt(); v.y = IO.ReadInt(); v.z = IO.ReadInt();
    return v;
  }

  public Vector CrossProduct(Vector that) {
    Vector c = new Vector();
    c.x = this.y * that.z - this.z * that.y;
    c.y = this.z * that.x - this.x * that.z;
    c.z = this.x * that.y - this.y * that.x;
    return c;
  }

  public int ScalarProduct(Vector that) {
    return this.x * that.x + this.y * that.y + this.z * that.z;
  }

  public bool Equals(Vector that) {
    return this.x == this.x && this.y == that.y && this.z == that.z;
  }
```

```
  public Vector Clone() {
    Vector v = new Vector();
    v.x = this.x; v.y = this.y; v.z = this.z;
    return v;
  }

  public void Write() {
    IO.Write("(");
    IO.Write(x, 1); IO.Write(","); IO.Write(y); IO.Write(","); IO.Write(z);
    IO.Write(")");
  }

}

class VecAlg {
// Demonstrate use of simple 3d vector algebra package
// P.D. Terry, Rhodes University, 2003

  public static void Main(String[] args) {
    Vector a = Vector.NewVector(1, 0, 0);
    Vector b = Vector.ReadVector();
    a.Write(); IO.Write(" x ");
    b.Write(); IO.Write(" = ");
    Vector c = a.CrossProduct(b);
    c.Write();
    IO.Write("\n");
    a.Write(); IO.Write(" . ");
    b.Write(); IO.Write(" = ");
    IO.Write(a.ScalarProduct(b));
    IO.Write(c.Equals(c.Clone()));
  }
}
```

The second example illustrates how a class may be defined for handling an expanding list of integers using the algorithm sometimes known as "amortized doubling".

```
  class IntList {
  // Expansible list of integers
  // After example on page 134 of
  // Gough: "Compiling for the .NET Common Language Runtime" (Prentice Hall, 2002)
    const int defaultHigh = 4;
    int high = defaultHigh, tide = 0;
    int[] list = new int[defaultHigh];

    public static IntList NewList(int size) {
    // pseudo constructor
      if (size <= 0) size = defaultHigh;
      IntList l = new IntList();
      l.high = size;
      l.list = new int[size];
      return l;
    }

    public void Write() {
      int i = 0;
      while (i < tide) {
        IO.Write(list[i]); i++;
      }
    }

    public void Add(int x) {
      if (tide == high) {
        int[] temp = list;
        high = 2 * high;
        list = new int[high];
        int i = 0;
        while (i < tide) {
          list[i] = temp[i];
          i++;
        }
      }
      list[tide] = x;
      tide++;
    }

  }
```

```
class ListDemo {
// Demonstrate use of expandable List class
// P.D. Terry, Rhodes University, 2003

  public static void Main(String[] args) {
    IntList myList = new IntList();
    // alternatively use something like myList = IntList.NewList(4);
    int i = 0;
    while (i < 4) {
      myList.Add(2*i + 1); i++;
    }
    myList.Write();
    IO.Write('\n');
    while (i < 14) {
      myList.Add(2*i + 1); i++;
    }
    myList.Write();
  }
}
```

17 IO library module

In the interests of simplicity, the implementations of C#Minor produce textual assembler code that has then to be assembled and linked with IO routines to produce a complete program.

The IO library handles input from "standard input" and generates output to "standard output", and should meet the minimum specification below. Code for such a library is supplied with the distribution kit. This code incorporates various other methods that may be of use in developing other applications.

```
class IO {
  public static int ReadInt()
  // Reads a textual representation of an integer and returns the value.  Leading spaces
  // are discarded.  Errors may be reported or quietly ignored (when 0 is returned).

  public static bool ReadBool()
  // Reads a word and returns a Boolean value, based on the first letter.
  // Typically the word would be T(rue) or Y(es) or F(alse) or N(o).

  public static char ReadChar()
  // Reads and returns a single character.

  public static string ReadString()
  // Reads and returns a string.  Incorporates leading white space, and returns
  // with a control character, which is not incorporated or consumed.

  public static string ReadLine()
  // Reads and returns a string.  Incorporates leading white space, and returns
  // when EOL or EOF is reached.  The EOL character is not incorporated
  // but is consumed.

  public static string ReadWord()
  // Reads and returns a word - a string delimited at either end by a control character
  // or space (typically the latter).  Leading spaces are discarded, and the
  // terminating character is not consumed.

  public static void Write(int x, int w)
  public static void Write(bool x, int w)
  public static void Write(char x, int w)
  public static void Write(string x, int w)
  // Writes a representation of the value of x to standard output.
  // If w = 0, x is preceded by exactly one space
  // If w > 0, x is written right justified in a field of at least w characters
  // If w < 0, x is written left justified in a field of at least -w characters

  public static void Write(int x)         { Write(x, 0); }
  public static void Write(bool x)        { Write(x, 0); }
  public static void Write(char x)        { Write(x, 1); }
  public static void Write(string x)      { Write(x, 1); }
}
```

18 Bibliography

Engel, J. (1999) *Programming for the Java Virtual Machine*, Addison-Wesley, Reading MA.

Mössenböck, H. (2004) http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/

Reiser, M. and Wirth, N. (1992) *Programming in Oberon*, Addison-Wesley, Wokingham, England.

Terry, P.D. (2005) *Compiling with C# and Java*, Pearson, London.

Wirth, N. (1985) *Programming in Modula-2* (3rd edn), Springer, Berlin.