The C ♭ (C 'flat') Language

Cb is based upon a language design named Z#, which was in turn based upon C#. The Z# language was designed by Hanspeter Mössenböck of the University of Linz specifically for use in a course on compiler construction. Z# has been modified to bring it closer again to C#, so close in fact that a C# compiler should accept Cb programs. All that is needed in addition to the Cb program is a precompiled library (a *dll*) named CbRuntime;

1 General Characteristics

- A Cb program consists of a single main class with a method. This class has no fields, and there are no other external classes.
- The main method of a Cb program is always called *Main( )*. When a Cb program is executed, this method is called by the operating system.
- Cb has
  - constants of type int (e.g. 123) and string (e.g. "abc").
  - variables: whose types are the basic types or the structured types listed below; these are *value types*, except for arrays which are implemented as *reference types*.
  - basic types: int, string
  - structured types: one-dimensional zero-based arrays, and structs.
  - one static method in the main class.
  - a predefined Length property for array and string values.

Sample Cb Program

```
using CbRuntime;

class P {

   public const int size = 10;

   public struct Table {
      public int[] pos;
      public int[] neg;
   }

   public static void Main( ) {
      Table val;
      int x, i;
      //---------- Initialize val ----------
      val.pos = new int[size];
      val.neg = new int[size];
      i = 0;
      while (i < size) {
         val.pos[i] = 0;
         val.neg[i] = 0;
         i++;
      }

      //---------- Read values ----------
      cbio.read(out x);
      while(-size < x && x < size) {
         if (0 <= x)
           val.pos[x]++;
         else
           val.neg[-x]++;
         cbio.read(out x);
      }
   }
}
```

2 Cb Language Syntax (in Extended BNF)

```
Program      = {"using" ident ";"}
               "class" ident "{" { ConstDecl | StructDecl | MethodDecl } "}".
ConstDecl    = "public" "const" Type ident "=" ( number | stringConst ) ";".
StructDecl   = "public" "struct" ident "{" { FieldDecl } "}".
FieldDecl    = "public" Type ident { "," ident } ";".
MethodDecl   = "public" "static" "void" ident "(" [ FormalPars ] ")" Block.
LocalDecl    = Type ident { "," ident } ";".
FormalPars   = FormalDecl { "," FormalDecl }.
FormalDecl   = Type ident.
Type         = ident [ "[" "]" ].
Statement    = Designator ( "=" Expr | "(" [ ActPars ] ")"  | "++" | "--" ) ";"
             | "if" "(" Condition ")" Statement [ "else" Statement ]
             | "while" "(" Condition ")" Statement
             | "break" ";"
             | "return" [ Expr ] ";"
             | ident "." ident "(" "out" Designator ")" ";"
             | ident "." ident "(" Expr ")" ";"
             | Block
             | ";".
Block        = "{" { LocalDecl  |  Statement } "}".
ActPars      = Expr { "," Expr }.
Condition    = CondTerm { "||" CondTerm }.
CondTerm     = CondFact { "&&" CondFact }.
CondFact     = EqFact  Eqop  EqFact.
EqFact       = Expr  RelOp  Expr.
Expr         = [ "-" ] Term { Addop Term }.
Term         = Factor { Mulop Factor }.
Factor       = Designator [ "(" [ ActPars ] ")" ]
             | number
             | stringConst [ “.” ident ]
             | "new" ident [ "[" Expr "]" ]
             | "(" Expr ")".
Designator   = ident { "." ident | "[" Expr "]" }
EqOp         = "==" | "!="
Relop        = ">" | ">=" | "<" | "<=".
Addop        = "+" | "-".
Mulop        = "*" | "/" | "%".
```

Lexical Structure

Keywords:	break, class, const, else, if, new, out, public, return, static, struct, using, void, while
Token classes:	ident = letter { letter | digit | "_" }. number = digit { digit }. stringConst = "" { printableChar | "\n" | "\r" } "".
Character sets:	letter = a .. z ∪ A .. Z digit = 0 .. 9 printableChar = letter ∪ digit ∪ ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~ (= ASCII: 32(' ') .. 126('~'))
Operators:	+ - * / % == != > >= < <= && || = ++ -- ; , . ( ) [ ] { }
Comments:	from /* to */ (these are nestable), and from // to the end of line.

3 Cb Semantics

All terms in this document that have a definition are underlined to emphasize their special meaning. The definitions of these terms are given here.

Reference type

Arrays are called reference types. A variable declared as an array type is implemented as a word of storage which contains a reference to a block of memory in the heap holding the array elements.

Type of a constant

- The type of an integer constant (e.g. 17) is int.
- The type of a string constant (e.g. "xyz") is string.

Same type

Two types are the same
- if they are denoted by the same type name, or
- if both types are arrays and their element types are the same.

Type compatibility

Two types are compatible
- if they are the same, or
- if one of them is a reference type and the other is the type of null.

Assignment compatibility

A type *src* is assignment compatible with a type *dst*
- if *src* and *dst* are the same, or

- if *dst* is a reference type and *src* is the type of null.

Predeclared names (i.e. these are identifiers, not keywords or reserved words)

int the type of all integer values
string the type of all string values
null the null value of a class or array variable, meaning "pointing to no value"
cbio a library class in the CbRuntime namespace containing the public static methods read and write which is overloaded to output int and string values.
Length a predefined property of array and string values.

Scope

A scope is the textual range of a method or a class. It extends from the point after the declaring method or class name to the closing curly brace of the method or class declaration. A scope excludes other scopes that are nested within it. We assume that there is an (artificial) outermost scope, to which the main class is local and which contains all predeclared names.
The declaration of a name in an inner scope S hides the declarations of the same name in outer scopes.

Notes

- Mutual recursion (i.e. method A calling method B and B calling A) is not allowed, since every name must be declared before it is used.
- A predeclared name (e.g. int or string) can be redeclared in an inner scope (but this is not recommended).
- The '++' and '--' operators are not supported in their full generality. There are only increment and decrement *statements* in Cb. These two operators are not available for use inside expressions.

4 Cb Context Conditions

General context conditions

- Every name must be declared before it is used.
- A name must not be declared twice in the same scope.
- A program must contain a method named 'Main'. It must be declared with a void function type and must not have parameters.

Context conditions for the Cb productions

Program = { "using" ident ";" }
"class" ident "{" { ConstDecl | StructDecl | MethodDecl } "}"

- Only one method may be declared and it must be named Main.
- If the using clause is provided, the name of the namespace which follows must be CbRuntime. Although the syntax permits several using clauses, only one namespace is currently provided.

ConstDecl = "public" "const" Type ident "=" (number | stringConst) ";"

- The type of *number* or *stringConst* must be the same as the type of *Type*.

FieldDecl = "public" Type ident { "," ident } ";"

StructDecl = "public" "struct" ident "{" { FieldDecl } "}"

- Following the declaration, *ident* denotes a new structured type. It is a value type.

MethodDecl = "public" "static" "void" ident "(" [FormalPars] ")" Block

- The method named Main must not have any formal parameters.

FormalPars = FormalDecl { "," FormalDecl }

FormalDecl = Type ident

Type = ident ["[" "]"]

- *ident* must denote a type. In Cb, the typenames are restricted to int, string or an identifier which has been declared as the name of a *struct*.

Statement = Designator "=" Expr ";"

- *Designator* must denote a variable, an array element or an object field.
- The type of *Expr* must be assignment compatible with the type of *Designator*.

Statement = Designator "(" [ActPars] ")" ";"

- *Designator* must denote a method, or
- *Designator* is the combination cbio.write and exactly one parameter of type int or string has been provided, and cbio refers to the name imported from the CbRuntime namespace.

Statement = Designator ("++" | "--") ";"

- *Designator* must denote a variable, an array element or an object field.
- *Designator* must be of type *int*.

Statement = "break" ";"

- The break statement must be contained in a loop statement (while).

Statement = "return" [Expr]

- The type of *Expr* must be assignable to the function type of the current method.
- If *Expr* is missing the current method must be declared as void.

Statement = Designator "(" "out" Designator ")" ";"

- The first *Designator* must be the combination cbio.read and cbio refers to the name imported from the CbRuntime namespace.
- The second *Designator* must denote a variable, an array element or an object field.
- The second *Designator* must be of type int.

Statement = ident "." ident "(" Expr ")" ";"

- *Expr* must be of type int or string.
- The two *ident* tokens must be the identifiers cbio and write, respectively, to refer to one of the static write methods in the cbio class contained in the CbRuntime namespace.

Statement = "if" "(" Condition ")" Statement ["else" Statement]
| "while" "(" Condition ")" Statement
| Block
| ";"

Block = "{" { LocalDecl | Statement } "}"

LocalDecl = Type ident { "," ident }

ActPars = Expr { "," Expr }

- The numbers of actual and formal parameters must match.
- The type of every actual parameter must be assignment compatible with the type of every formal parameter at corresponding positions.

Condition = CondTerm { "||" CondTerm }

CondTerm = CondFact { "&&" CondFact }

CondFact = Expr Relop Expr

- The types of both expressions must be compatible.
- Classes and arrays can only be checked for equality or inequality.

Expr = Term

Expr = "-" Term

- *Term* must be of type int.

Expr = Expr Addop Term

- *Expr* and *Term* must be of type int.

Term = Factor

Term = Term Mulop Factor

- *Term* and *Factor* must be of type int.

Factor = Designator | number | "(" Expr ")"

Factor = stringConst [“.” ident]

- If *ident* is provided, it must be the identifier “Length”.

Factor = Designator "(" [ActPars] ")"

- *Designator* must denote a method whose type must not be void.

Factor = "new" Type

- *Type* must denote an inner class.

Factor = "new" Type "[" Expr "]"

- The type of *Expr* must be int.

Designator = Designator "." ident

- The type of *Designator* must be a struct or it must be the class cbio imported from CbRuntime.
- *ident* must be a field or method of *Designator*.

Designator = Designator "[" Expr "]"

- The type of *Designator* must be an array.
- The type of *Expr* must be int.

Relop = "==" | "!=" | ">" | ">=" | "<" | "<="

Addop = "+" | "-"

Mulop = "*" | "/" | "%"

5 Implementation Restrictions

- A method must not have more than 256 local variables.
- A method must not have more than 256 arguments.