The C♭ (C 'flat') Language for 2014

From this point on, C♭ will be written as Cb. (Much less trouble to type!)

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. For csc435/535 in 2014, we have modified the language to bring it so close to C# that a C# compiler should accept Cb programs.

1 General Characteristics

- A Cb program consists of an optional `using` declaration followed by one or more class declarations. The only namespace which may be referenced in the `using` declaration is `System`.
- Each class contains members, all of which are declared as `public`.
- A Cb class possesses only a default constructor. (It causes all instance fields to be initialized to 0 or `null`, as appropriate.)
- A class may inherit from a parent class. If no parent class is specified, the parent class is implicitly `Object`.
- The members of a class can be declarations of static fields used as constants (they must be initialized with a constant in the declaration), declarations of instance variables (i.e. non-static fields) which are not explicitly initialized, or declarations of methods. Both static and non-static methods are available.
- Exactly one class of the Cb program must contain a static method named `Main` which has no formal parameters. When a Cb program is executed, this method is called by the operating system.
- Methods may have a `void` type or may return a value of any type in the Cb language (but not Boolean values). Methods may accept zero or more formal parameters which are passed by value. (The `ref` and `out` parameter passing mechanisms of C# are not supported.)
- Method overloading (two methods with the same name in a class with numbers or types of formal parameters) is permitted.
- All non-static methods in a Cb program are *virtual*. Method overriding in a subclass is permitted.
- Fields, method return values, formal parameters of methods and local variables in methods may be declared to have any of the following types:
  - `int`, `char`, `string`
  - any class declared in the Cb program
  - a one-dimensional array of any of the types listed here.

  Note that Boolean values exist in a Cb program (they are generated by comparisons and tested by *if* and *while* statements), but these values cannot be saved in variables or passed to methods.
- Statements take one of the forms
  - assignment
  - an expression executed for its side-effects (this is a method invocation or a use of `++` or `--`)
  - *if*-statement (with optional *else* clause)
  - *while* loop

- `return` (with an optional result expression)
- `break`.

- Expressions may perform these operations when evaluated
  - integer operations of `+ - * / %` (`+` and `-` may also be used as prefix operators);
  - string concatenation using the + operator;
  - `++` or `--` used as postincrement operators, and only when the expression is used as a statement;
  - comparison operations between integers: `== != <= < > >=`
  - array indexing;
  - string indexing (i.e. `s[i]` returns the *i*-th character of string *s*);
  - obtain the length of a string or array using the `Length` property;
  - extract a substring of a string value using the `Substring` method;
  - convert a string to an int value using the `Int32.Parse` method.
- The only usage of library classes in a Cb program which is supported takes these forms
  - `System.Console.Write(e)` or `System.Console.Write(e)`, where e has `string` or `int` or `char` type;
  - `System.Console.WriteLine()` or `System.Console.WriteLine(e)` where *e* is `int` or `char` or `string`;
  - `System.Console.ReadLine()`;
  - s`.Substring(i)` or s`.Substring(i,j)` where s has `string` type;
  - e`.Length` where *e* has a `string` or a one-dimensional array type;
  - `System.Int32.Parse(s)` where *s* is a `string` (to be converted to an `int` value).

Important Note

The Cb language is required to be a strict subset of the C# language. If an input file is rejected by the C# compiler then it must be rejected by the Cb compiler too. If an input file is accepted by the Cb compiler then it must also be accepted by the C# compiler.

(It should not matter which version of C# is being used to test this subset requirement; Cb is meant to be a strict subset of C# 2.0.)

If there is a discrepancy between this document and the Microsoft C# compiler (for C# 4.5) where the C# compiler rejects a program that appears to be allowed by the explantions and syntax rules given here, then the rules enforced by the compiler should have precedence. (Please report any such discrepancies so that this document can be amended.)

A Sample Cb Program

```
/* Linked-list example */

using System;

class List {
  public List next;

  public virtual void Print() {}
}

class Other: List {
  public char c;

  public override void Print() {
    Console.Write(' ');
    Console.Write(c);
    if (next != null)
      next.Print();
  }
}


class Digit: List {
  public int d;

  public override void Print() {
    Console.Write(' ');
    Console.Write(d);
    if (next != null)
      next.Print();
  }
}
```

```
class Lists {
  public static void Main() {
    List ccc;
    string s;
    Console.WriteLine(
           "enter some text => ");
    s = Console.ReadLine();
    ccc = null;
    int i;
    i = 0;
    while(i < s.Length) {
      char ch;
      ch = s[i];  i++;
      List elem;
      if (ch>='0' && ch<='9') {
        Digit elemD;
        elemD = new Digit();
        elemD.d = ch - '0';
        elem = elemD;
      } else {
        Other elemO;
        elemO = new Other();
        elemO.c = ch;
        elem = elemO;
      }
      elem.next = ccc;
      ccc = elem;
    }
    Console.WriteLine(
        "\nReversed text =");
    ccc.Print();
    Console.WriteLine("\n");
  }
}
```

This sample program is provided separately as the file `CbExample.cs`.

2 Cb Language Syntax (in Extended BNF)

Program	=	{ "using" ident ";" } ClassDecl { ClassDecl }
ClassDecls	=	"class" ident [":" ident] "{" { MemberDecl } "}"
MemberDecl	=	ConstDecl | FieldDecl | MethodDecl
ConstDecl	=	"public" "const" Type ident "=" (intConst | stringConst | charConst) ";"
FieldDecl	=	"public" Type ident { "," ident } ";"
MethodDecl	=	"public" ("static" | "virtual" | "override") ("void"| Type) ident "(" { FormalPars } ")" Block
LocalDecl	=	Type ident { "," ident } ";"
FormalPars	=	FormalDecl { "," FormalDecl }
FormalDecl	=	Type ident
Type	=	(ident | "int" | "string" | "char") ["[" "]"]
Statement	=	Designator "=" Expr ";"
	|	"if" "(" Condition ")" Statement ["else" Statement]
	|	"while" "(" Condition ")" Statement
	|	"break" ";"
	|	"return" [Expr] ";"
	|	Designator "(" ActualPars ")" ";"
	|	Designator ("++" | "--") ";"
	|	Block
	|	";"
Block	=	"{" { LocalDecl | Statement } "}"
ActPars	=	Expr { "," Expr }
Condition	=	CondTerm { "||" CondTerm }
CondTerm	=	CondFact { "&&" CondFact }
CondFact	=	EqFact Eqop EqFact
EqFact	=	Expr RelOp Expr
	=	["+" | "-"] Term { Addop Term }
Term	=	Factor { Mulop Factor }

Factor	=	Designator ["(" [ActPars] ")"]
	|	intConst
	|	charConst
	|	stringConst ["." ident]
	|	"new" (ident | "int" | "string" | "char") "[" Expr "]"
	|	"new" ident "(" ")"
	|	"null"
	|	"(" Expr ")" Factor [a]
	|	"(" NonClassType ")" Factor
	|	"(" Expr ")"
Designator	=	ident { "." ident | "[" Expr "]" }
EqOp	=	"==" | "!="
Relop	=	">" | ">=" | "<" | "<="
Addop	=	"+" | "-"
Mulop	=	"*" | "/" | "%"
NonClass-Type	=	("int" | "string" | "char") ["[" "]"]

a.Note: this rule should have been written as Factor = "(" Type ")" Factor , it is the rule for casting a value to a diferent datatype. However, the desired form of the rule introduces shift-reduce conflicts in the parser unless the grammar is extensively rewritten. The form shown in the table accepts some nonsense for a cast, and there will need to be a later semantic check in the compiler that the quantity between the parentheses is a class name.

Some Lexical Structure Details

- Identifiers begin with a letter in the ASCII subset (i.e. ['a' .. 'z', 'A' .. 'Z']) and are followed by zero or more letters or digits or underscores (again, only letters in the ASCII subset may be used).
- Integer constants are comprised of one or more decimal digits. (A leading '0' digit does not have special meaning as it does in C/C++ or Java).
- String constants can contain any printable character in the ASCII subset plus space characters and these five escape combinations: `"\r" "\n" "\t" "\"" "\'"`.
- Character constants contain a single instance of any character permitted in a string constant.
- Both the `//` and `/*...*/` forms of comment may be used. The latter form of comment can be nested – even though is not permitted by the Microsoft C# compiler!

Some Semantic Details

Reference type

Arrays and classes are called *reference types*. A variable declared with a reference type is implemented as a word of storage which contains a reference to a block of memory in the heap holding the array or class instance.

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.

Assignment compatibility

A type *src* is assignment compatible with a type *dst* if (1) *src* and *dst* have the same type, or (2) if *dst* has the `int` type and *src* has the `char` type, or (3) if *dst* is a reference type and *src* is either the `null` constant or is a subclass of *dst*.

Restriction: if *src* and *dst* both have array types and their element types are reference types, the two reference types must be identical. (This restriction does not exist in Java or C# which support array covariance.)

Comparison compatibility

Two values with types t1 and t2 may be compared for equality (or inequality) if t1 and t2 have the same type, or if one type is assignment compatible with the other type (either way around), or if both values are `null`.

Two values may be compared according to an ordering relation only if the two values have `int` or `char` types.

Miscellaneous Restrictions

- If the `using` clause is provided, the name of the namespace which follows must be `System`. Although the syntax permits several `using` clauses, only one namespace is currently provided.
- In an assignment statement and the increment or decrement operations,

  Statement = Designator "=" Expr ";"
  Statement = Designator "++" ";"
  Statement = Designator "--" ";”

  Designator must denote a local variable, a formal parameter, an array element or a field of a class instance. For an assignment, the type of *Expr* must be assignment compatible with the type of *Designator*. For an increment/decrement operation, the type of *Designator* must be `int` or `char`.
- A `break` statement must be contained in a loop statement (`while`).
- If a return statement returns a value, that variable must be assignment compatible with the declared result type of the containing method.
- Array indexes must have `int` or `char` type.

5 Implementation Restrictions

The Cb compiler can assume that:

- no method will contain more than 255 local variables;
- no method will contain more than 255 formal parameters;
- no program contains more than 255 methods across all the declared classes;
- no class contains more than 255 fields.