BC# is a (nontrivial) simplification of the C# language. The following is its description. Like many such brief descriptions, it may be ambiguous or incomplete. In such cases, use your best judgment and document your decisions.

1 Lexical Issues

• **Identifiers**: An identifier starts with a letter or underscore, and then can consist of any number of letters, underscores, or digits. Case is significant when comparing identifiers.

• **Comments**: A comment is either a line comment, beginning with // and going to the end of the line, or a delimited comment, starting with /* and ending with the next */. Comments do not nest: The character sequences */ and // have no special meaning within a comment.

• **Whitespace**: White space and indentation (and comments) do not affect the meaning of the program, except to separate tokens. Whitespace characters include space, newline, carriage return, tab, form-feed, and vertical tab, sometimes written\(^1\) " \n\r\t\f\v".

Information on other tokens and keywords is interspersed throughout the rest of this document.

2 Programs

A program is a collection of class definitions. Exactly one of the classes must be named `Start`, and this class must contain a member `static void Main` whose argument is an array of strings.\(^2\)

The names of all classes must be distinct, and the names of all fields and methods within a single class must be distinct. In particular, BC# does not support overloading.

Classes are all defined mutually-recursively (any class can refer to any other class in the file without needing forward declarations).

\(^1\)\texttt{ml-ulex} understands all these escape codes, but \texttt{ml-lex} doesn’t.

\(^2\)It is probably not a good idea to enforce these constraints in the parser.
Data types

The types of BC# are:

1. **int**. Integer constants can be written either using base ten digits, or using hexadecimal digits (preceded by \(0x\) or \(0X\)).

2. **char**. A single ASCII character. Written as a single character in single quotes, e.g., ‘a’ or ‘\n’.

3. **bool**. The two constants of this type are false and true. (Both constants are reserved words.)

4. **string**. String constants are delimited by double quotes, and must be on a single line. String constants can contain escape codes (e.g., \(\text{n}\) to represent the newline character, or \(\text{\"}\), which is a double quote character rather than the end-of-string marker).

   Alternatively, a string constant can be written @"..." to mean a literal string with no expansion of escape codes. Such strings can even span multiple lines (in which case the string includes newline characters), e.g.,
   
   ```
   string s = @"Line1
   Line2\n\n\n";
   ```

   defines a string containing a single newline characters, five occurrences of the letter n, and three backslashes, while
   
   ```
   string s = "Line1\nLine2\n\n\n";
   ```

   defines a string with four newline characters, two actual n’s and no backslashes.

5. Array types (**t[]** for any type **t**).

6. Class types. These include the built-in type **object** (which can also be written **Object**), and any user-defined class types, all of which (eventually) inherit from **object**.

All the above-named built-in types (e.g., **int** and **object**) are reserved words.

The only implicit conversions are

- From **char** to **int**.

---

\(^3\)This is zero-x, not the word ox.

The function **StringCvt.scanString (Int.scan StringCvt.DEC) : string -> int option** can convert decimal strings to integers. Replace **DEC** by **HEX** for hexadecimal strings.

\(^4\)C# actually uses Unicode.

\(^5\)It might be relevant here that . in **ml-lex** does not match newline characters, but . in **ml-ulex** does.

\(^6\)Hint: the SML function **String.fromCString : string -> string option** converts any embedded escape codes into the characters they represent. The function **String.toCString : string -> string** goes the other way. **Char.fromCString** and **Char.toCString** work similarly.
• From any class to a class that it inherits from.

Note that there is no conversion from int to bool or vice-versa, no conversions among array types, and no conversion between non-class types and object.

3 Expressions

Every expression has a type, which describes the sort of value it returns. Unless otherwise mentioned, operators in BC# have the same precedence and associativity as in C#.

• Constants and identifiers.
• Unary (-, ++, --) and binary (+, -, *, /) arithmetic operators.
• Arithmetic, string, and character comparisons (<, <=, >, >=) returning a boolean.
• Unary (!) and [short-circuiting] binary (&&, ||) logical operators.
• Binary equality (==) and inequality (!=), for any two values of the same type. This implements integer equality, string equality, or address-equality, depending on the type of the arguments.
• Conditional expression (e1 ? e2 : e3). The values of e2 and e3 must be convertible to a common type.
• Assignment (lvalue = rvalue). The value stored is also the value of the whole assignment operation. Assignment is right-associative.
• Binary string concatenation (s1 + s2).
• Array indexing (e[n]), or extracting a character from a string (e[n]) to obtain a character.
• Values can be created by (new t(...) for any class type t. This creates a new object and calls the constructor with the given arguments.
• Arrays can be created by new t[e][...][] to create an array of values of type t[...][] of length e, where again t must be a non-array type. (E.g., new int[3] to create an array of three integers, or new int[3][] to create an array of three integer-arrays.) The elements of this array are initially zero, false, or null as appropriate.
• The constant null, which can be treated as having any class type, array type, or type string.
• The constant this can be used in non-static methods and constructors to refer to the enclosing object.
• $e.l$ extracts the value of the field $l$ from the object resulting from evaluating $e$. Alternatively, $X.l$ extracts the value of the static field $l$ from the class $X$.\(^7\)

• $e.m(...)\text{ is invocation of the virtual method } m \text{ in the object resulting from evaluating } e$. The ... must be a collection of comma-separated arguments in parentheses. Each argument an expression passed by value. Alternatively, $X.m(...)\text{ invokes the static method } m \text{ in the class } X$.\(^8\)

• $e.toString()\text{ returns the value of the expression } e \text{ expressed as a string. If } e \text{ is an integer you get a string representation of the integer. If it is a character you get a 1-character string. If it is a boolean you get the string "true" or "false". If it is a string, you get the same string back. If it is a class, you get the result of the class's } toString \text{ method, which is a virtual method inherited from the object (that can be overwritten).}$

• $e.Length\text{ returns the number of characters in the string } e$.

• $Console.In.readLine()\text{ gets a line of input from the standard input and returns it as a string.}$

4 \textbf{L-values}

An l-value is something that can appear on the left-hand-side of an assignment statement. Permissible l-values are:

• Identifiers

• Array accesses $e_1[e_2]$

• static fields $X.f$, or object fields $e.f$.

Strings are immutable, although variables of type \texttt{string} can be assigned different (immutable) string values.

5 \textbf{Statements}

Statements are executed for their side effects, and do not return any values.

• The empty statement $; \text{ does nothing.}$

• If $e \text{ is an expression, then } e ; \text{ is a statement that evaluates } e \text{ and throws away the result.}$

\(^7\)Don’t try to distinguish the two during parsing.  
\(^8\)Ditto.
• A block statement is a sequence of zero or more statements enclosed in { and }.

• if (e) stmt1 and if (e) stmt1 else stmt2 are statements if e is a boolean expression, and stmt1 and stmt2 are statements. The parentheses are required.

• while (e) stmt is a statement if e is a boolean expression, and stmt is a statement. The parentheses are required.

• for (e1; e2; e3) stmt is the usual loop structure.\(^9\)

• return; and return e; are statements; the former is used in void functions, and the latter is used if we need to return the value of the expression e.

• A variable declaration t x = e; defines a variable x of type t and initial value e. The scope of the variable is the rest of the enclosing statement.

• Console.write(e); is a statement for any expression e. It displays the value of e.toString() to the standard output stream.

6 Classes

A class declaration has the form

```
class identifier : identifier {
  optional-fields
  required-constructor
  optional-methods
}
```

where the first identifier is the class name, and then after the colon is the user-defined class being inherited from. (One cannot inherit from the other built-in types, e.g., int or string.) The colon and the superclass can be omitted, in which case the class implicitly inherits from object. The semicolon at the end is also optional.

Each field may be static or not. static fields have a type and an initial value; non-static fields just have a type. All BC# fields are (implicitly) public.

The constructor must have the form

```
Classname (parameters) : base(superclass-constructor-arguments) { body }
```

The arguments to the superclass constructor can be arbitrary expressions.

A method may be static, virtual, or override (exactly one of the three — override means the same as virtual except that a method of the same name with the same parameter and return types must have been inherited). It must then have a return type or be declared as returning void, have a name, and have a parenthesized, comma-separated list of formal parameters. The body of a method is a block statement. Within a method or constructor, one must always use this to refer to methods and fields in the surrounding object.

\(^9\)Note that e1 must be an expression and not a variable declaration.
Example

The following code gives a brief example of a BC# program:

class Point : object {
    static int points_created = 0; // Initializer required
    int x; // Initializer forbidden
    int y;

    Point(int x0, int y0) : base() // object has a zero-argument constructor
    {
        this.x = x0; // Recall that the reference to this is required
        this.y = y0; // Ditto
        ++Point.points_created; // Explicit class reference required
    }

    virtual void move(int dx, int dy) {
        this.x = this.x + dx;
        this.y = this.y + dy;
        return;
    }
};

class VPoint : Point {
    bool visible;
    VPoint (int x1, int y1, bool isVisible) : base(x1, y1) {
        this.visible = isVisible;
        return;
    }
};

class Main : object {
    Main() : base() { return; } // Every class must have a constructor

    static void Main(string[] args)
    {
        Point myPoint = new VPoint(7,3,true);
        myPoint.move(3,4);
        myPoint.move('a',-0xFf);
        Console.write(Point.points_created.toString() + "\n");
        Console.write(myPoint.x);
        Console.write(" " + myPoint.y.toString() + "\n");
        return;
    }
};