1. 介绍

MY-BASIC 是一个轻量级的BASIC解释器，用标准C语言编写成双文件形式。它旨在具有嵌入性、可扩展性和可移植性。它是一种动态类型的编程语言，保留了结构化语法，支持一种基于原型的编程风格（面向对象编程），同时通过lambda抽象实现了函数式编程范式。核心部分由一个C源文件和一个相关的头文件组成。您可以将其作为独立的解释器使用，也可以将其嵌入到现有的C、C++、Java、Objective-C、Swift、C#等项目中，通过添加您自己的脚本接口进行完全定制。您还可以学习如何从零开始构建一个解释器，或者基于它构建自己的方言。

这份手册是关于如何使用MY-BASIC进行编程的快速参考，介绍了它的功能、使用方法以及如何将其扩展为一种脚本编程语言。

欲了解最新修订版本或其他信息，请访问 https://github.com/paladin-t/my_basic。

2. 使用BASIC进行编程

众所周知，编程语言BASIC是"初学者通用符号指令代码"的缩写；当我们今天提到BASIC时，通常是指BASIC家族，而不是特定的一个。BASIC家族有着悠久的历史，最早的BASIC是由约翰·乔治·凯门伊（John George Kemeny）和托马斯·尤金·库尔茨（Thomas Eugene Kurtz）于1964年在新罕布什尔州达特茅斯学院设计的；BASIC因其易于学习和使用而一直享有盛誉。感谢所有致力于BASIC的人和狂热爱好者，这种激情影响了很多像我这样的人，引导我们进入了计算机世界。

MY-BASIC提供了结构化的BASIC语法，以及许多其他复古和现代特性。如果您以前曾使用过其他BASIC方言或其他编程语言进行编程，您会对它感到熟悉。

入门指南
您可以从 https://github.com/paladin-t/my_basic/archive/master.zip 下载最新的MY-BASIC软件包，或者手动构建代码库。您可以通过 git clone https://github.com/paladin-t/my_basic.git 获取最新版本。在这一部分，让我们开始使用MY-BASIC解释器，如下所示：
闭合方括号是一个输入提示符。从一个 "Hello World" 惯例开始使用MY-BASIC：

```
'Hello world tutorial
input "What is your name: , n$
def greeting(a, b)
return a + " " + b + " by " + n$ + "."
enddef
print greeting("Hello", "world");
```

与其他BASIC方言一样，MY-BASIC不区分大小写；换句话说，PRINT A$ 和 Print a$ 的含义完全相同。在输入上述代码后，您将获得一个问候文本，然后输入一个RUN命令并按下回车键。以单引号开始的任何文本，直到该行结束，都是注释（也称为备注），不会影响程序的逻辑；注释不执行任何操作，只是代码的简短解释。您也可以使用经典的REM语句来开始注释。
MY-BASIC可以通过宏进行配置，本手册是基于默认配置编写的。

多行注释
MY-BASIC 还支持多行注释，这与其他方言相比是一个小优势。多行注释被 *'[and ']* 包围，例如：

```
print 'Begin';
'[
print "Thisline won't be executed!";
print "This is also ignored";
']
print "End";
```

MY-BASIC 会忽略这两者之间的所有注释行。只需将 '[修改为 "[就可以取消所有行的注释，非常方便。

对Unicode的支持
Unicode如今广泛用于国际文本表示；MY-BASIC支持基于Unicode的标识符和字符串操作。例如：

```
print "你好" + "世界";
日本語 = "こんにちは"
print 日本語, ", ", len(日本語);
```

关键字
在MY-BASIC中有一些关键字和保留函数，如下所示：

Keywords	REM, NIL, MOD, AND, OR, NOT, IS, LET, DIM, IF, THEN, ELSEIF, ELSE, ENDIF, FOR, IN, TO, STEP, NEXT, WHILE, WEND, DO, UNTIL, EXIT, GOTO, GOSUB, RETURN, CALL, DEF, ENDDEF, CLASS, ENDCLASS, ME, NEW, VAR, REFLECT, LAMBDA, MEM, TYPE, IMPORT, END
Reserved functions Standard library	ABS, SGN, SQR, FLOOR, CEIL, FIX, ROUND, SRND, RND, SIN, COS, TAN, ASIN, ACOS, ATAN, EXP, LOG, ASC, CHR, LEFT, LEN, MID, RIGHT, STR, VAL, PRINT, INPUT
Collection library	LIST, DICT, PUSH, POP, BACK, INSERT, SORT, EXISTS, INDEX_OF, GET, SET, REMOVE, CLEAR, CLONE, TO_ARRAY, ITERATOR, MOVE_NEXT

在用户定义的标识符中不允许使用这些词；此外，还有两个预定义的布尔常量，即TRUE和FALSE，如字面意思所示，表示布尔值true和false，不允许将这些符号重新赋值为其他值。关键字和函数的详细内容将在本手册的后面进行解释。

运算符
MY-BASIC 中的所有运算符如下所示：

Operators	+, -, *, /, ^, =, <, >, <=, >=, <>

这些运算符用于计算或比较表达式。此外，关键字 MOD、AND、OR、NOT、IS 也可以作为运算符使用。表达式从左到右进行评估，优先级从上到下如下所示：

Level	Operation
1	( ) (explicit priority indicator)

2	- (negative), NOT
3	^
4	*, /, MOD
5	+, - (minus)
6	<, >, <=, >=, <>, = (equal comparison)
7	AND, OR, IS
8	= (assignment)

MOD 代表模数运算，即在其他某些编程语言中表示为 %。插入符号 ^ 代表幂运算。

数据类型和操作
MY-BASIC 是一种动态编程语言，因此变量没有类型，但值有类型。内置类型包括：Nil、整数、实数、字符串、类型、数组、列表、列表迭代器、字典、字典迭代器、原型（也称为“类”）和子例程（包括 Lambda）。此外，MY-BASIC 还支持用户自定义数据类型（用户类型和引用用户类型），以定制您自己的数据结构。

Nil 是一种特殊类型，只包括一个有效值 NIL，也称为null、none、nothing等。通过将变量赋值为nil，它可以处理/取消引用变量的先前值。

Type 类型的值表示一个值的类型，它将在 TYPE 语句中进行解释。

整数和实数被定义为 int 和 float C 类型，在大多数编译器下都是32位长。您可以通过修改几行代码来将它们重新定义为其他类型，比如 long、long long、double 和 long double。由于没有专门的布尔类型，布尔值也被定义为整数，并且可以从任何表达式赋值得到。布尔表达式的结果为 false 当为 NIL、FALSE 和 0 时；当为所有其他值，包括空字符串 "" 时，结果为 true。

MY-BASIC 接受十六进制和八进制的文字数字。十六进制数以 0x 前缀开始，八进制数以 0 开头。例如，0x10（HEX）等于 020（OCT）等于 16（DEC）。

变量标识符由字母、数字、下划线和可选的美元符号后缀组成，但必须以字母或下划线开头。不需要为变量使用类型说明符，并且在使用之前无需声明。通常情况下，不需要手动在整数和浮点数之间进行转换，MY-BASIC 会自动存储适当的数据类型。美元符号 $ 是从传统BASIC方言中保留下来的，作为变量标识符的有效后缀。但在大多数情况下，它并不表示字符串的类型。然而，DIM 和 INPUT 语句有时会特殊处理 $ 符号。赋值语句由一个可选的起始关键字 LET 和随后的赋值表达式组成。例如：

```
let a = 1 ' Assignment statement begins with LET
pi = 3.14 ' Another assignment statement without LET
```

MY-BASIC 默认支持最多四维的数组，这是通过宏定义的。数组是编程中一种常见的集合数据结构。数组可以存储一组数据，每个元素可以通过数组名和下标访问。在使用数组之前，必须通过 DIM（维度的缩写）语句进行声明，例如：

```
dim nums(10)
dim strs$(2, 5)
```

在MY-BASIC中，数组的常规命名规则与变量命名规则相同，实际上，MY-BASIC中的所有用户标识符都遵循相同的规则。数组可以是实数或字符串值的集合，这取决于标识符是否以 $ 符号结尾。数组的维度由逗号分隔。在MY-BASIC中，数组索引从零开始，因此 nums(0) 是数组 nums 的第一个元素，这与其他BASIC有些不同，但在现代编程语言中更为常见。数组索引可以是一个由常量、整数变量或结果为整数的表达式形成的非负整数值；无效的索引可能会导致越界错误。

可以使用加号运算符 + 将两个字符串连接在一起。每次字符串连接都会生成一个新的字符串对象并进行内存分配。还可以对字符串应用比较运算符，它从字符串的第一个字符开始进行比较，如果它们相等，就会继续检查后面的字符，直到出现差异或达到任何一个字符串的末尾；然后返回一个表示差异的整数值。

结构化例程
可以使用子例程提取可重用的代码块。MY-BASIC支持使用CALL/DEF/ENDDEF语句进行结构化例程和使用GOSUB/RETURN（和GOTO）语句进行指令式例程，但它们不能在同一个程序中混合使用。

结构化例程以DEF语句开头，以ENDDEF结尾，可以定义任意参数数目的例程。调用子例程类似于调用本地脚本接口。如果例程在调用之后进行词法定义，则需要使用显式的CALL语句。例程将最后一个表达式的值返回给调用者，或者使用RETURN语句进行显式返回。例如：

```
a = 1
b = 0

def fun(d)
    d = call bar(d)
    sin(10)
    return d ' 尝试注释这一行
enddef

def foo(b)
    a = 2
    return a + b
enddef

def bar(c)
    return foo(c)
enddef

r = fun(2 * 5)
print r; a; b;
```

每个例程都有自己的变量查找范围。
此外，CALL() 语句用于获取可调用的值，如下所示：

```
routine = call(fun) ' 获取可调用值
routine() ' 调用可调用值
```

请注意，获取值需要一对括号，否则将进行调用执行。

指令式例程
传统的指令式例程也得以保留。标签用于标记指令式例程的起始点。在程序中的任何地方使用 GOSUB 语句调用被标记的例程。RETURN 语句用于退出例程并将控制权返回给其调用者。

控制结构
在MY-BASIC中有三种执行流程。

串行结构是最基本的一种，逐行执行语句。MY-BASIC 支持 GOTO 语句，它提供了无条件的控制转移能力。使用方式类似于 GOSUB，但是使用 GOTO 标签。区别在于指令式例程可以从被调用者返回，而无条件的 GOTO 不能。END 语句可以放置在源代码的任何位置，用于终止整个程序的执行。

条件结构由一些条件跳转语句组成：IF/THEN/ELSEIF/ELSE/ENDIF。这些语句检查条件表达式，然后在条件为 true 时执行某个操作，在条件为 false 时执行其他操作，正如您编写的那样。

在单行中使用条件 IF 语句：

```
if n MOD 2 THEN PRINT "Odd" ELSE PRINT "Even"
```

或者多行形式：

```
INPUT n
IF n = 1 THEN
    PRINT "One";
ELSEIF n = 2 THEN
    PRINT "Two";
ELSEIF n = 3 THEN
    PRINT "Three";
ELSE
    PRINT "More than that";
ENDIF
```

它支持多行条件语句的嵌套 IF。请注意，单行 IF 不需要 ENDIF，而多行 IF 则需要。
循环结构语句会检查循环条件，如果条件为 true，就会执行循环体，直到条件为 false。

使用 FOR/TO/STEP/NEXT 语句来循环特定次数。例如：

```
FOR i = 1 TO 10 STEP 1
    PRINT i;
NEXT i
```

如果增量为1，那么STEP部分是可选的。在NEXT之后的循环变量也是可选的，如果它与相应的FOR有关。MY-BASIC 还支持使用 FOR/IN/NEXT 语句对集合进行循环。可以迭代列表、字典、可迭代类和用户类型。循环变量被赋予迭代器当前指向的元素的值。例如，在列表中进行从一到五的计数：

```
FOR i IN LIST(1 TO 5)
    PRINT i;
NEXT
```

WHILE/WEND 循环和 DO/UNTIL 循环用于在不确定的步骤中进行循环，或等待特定条件。例如：

```
a = 1
while a ≤ 10
print a;
a = a + 1
wend
```

```
a = 1
do
print a;
a = a + 1
until a > 10
```

正如它们的名称所示，WHILE/WEND 语句在条件为真时执行循环体，而 DO/UNTIL 语句则在条件为假之前执行循环体。WHILE/WEND 语句在执行循环体之前检查条件，而 DO/UNTIL 语句在执行循环体一次后检查条件。EXIT 语句中断当前循环，并继续执行位于循环外部的程序。

使用类

MY-BASIC 支持基于原型的面向对象编程（OOP，Object-Oriented Programming）。它也被称为“原型式”、“原型导向”、“无类”或“基于实例”的编程。

使用一对 CLASS/ENDCLASS 语句来定义原型对象（类）。使用 VAR 在类中声明成员变量。还可以使用 DEF/ENDDEF 语句在原型中定义成员函数（也称为“方法”）。在声明语句之后，使用一对括号将另一个原型括起来，表示从它继承（即将其用作元类）。使用 NEW 语句创建原型的新实例。例如：

```
CLASS Foo
    VAR a = 1
    DEF fun(b)
        RETURN a + b
    END DEF
END CLASS

CLASS Bar(Foo) ' 使用 Foo 作为元类（继承）
    VAR a = 2
END CLASS

inst = NEW(Bar) ' 创建 Bar 的新实例
PRINT inst.fun(3);
```

bar 仅将 foo 作为元类链接。inst 创建了 bar 的新克隆，并保留了 foo 的元链接。

MY-BASIC 支持使用 REFLECT 语句对原型进行反射。它迭代类和其元类中的所有变量字段和子例程，并将变量的名称/值对和子例程的名称/类型对存储到字典中。例如：

```
CLASS Base
    VAR b = "Base"
    DEF fun()
        PRINT b;
    END DEF
END CLASS

CLASS Derived(Base)
    VAR d = "Derived"
    DEF fun()
        PRINT d;
    END DEF
END CLASS

i = NEW(Derived)
i.fun()
```

```
r = REFLECT(i)
f = ITERATOR(r)
WHILE MOVE_NEXT(f)
    k = GET(f)
    v = r(k)
    PRINT k, ": ", v;
WEND

g = GET(i, "fun")
g()
```

使用 Lambda
Lambda 抽象（也称为“匿名函数”或“函数文字”）是一个没有绑定到标识符的函数定义。Lambda 函数通常用于以下情况：

1. 作为传递给其他函数的参数，或
2. 作为函数的返回值。
   Lambda 在捕获了外部作用域中的一些值后成为闭包。

MY-BASIC 对 lambda 提供了完全支持，包括作为值可调用、高阶函数、闭包和柯里化等等。

Lambda 表达式以 LAMBDA 关键字开始。例如：

```
' 简单调用
f = LAMBDA (x, y) (RETURN x * x + y * y)
PRINT f(3, 4);
```

```
' 高阶函数
DEF foo()
    y = 1
    RETURN LAMBDA (x, z) (RETURN x + y + z)
END DEF

l = foo()
PRINT l(2, 3);
```

```
' 闭包
s = 0
DEF create_lambda()
    v = 0
    RETURN LAMBDA ()
        (
            v = v + 1
            s = s + 1
            PRINT v;
            PRINT s;
        )
END DEF

a = create_lambda()
b = create_lambda()
a()
```

```
b()
```

```
' 柯里化
DEF divide(x, y)
    RETURN x / y
END DEF

DEF divisor(d)
    RETURN LAMBDA (x) (RETURN divide(x, d))
END DEF

half = divisor(2)
third = divisor(3)
PRINT half(32); third(32);
```

```
' 作为返回值
DEF counter()
    c = 0
    RETURN LAMBDA (n)
        (
            c = c + n
            PRINT c;
        )
END DEF

acc = counter()
acc(1)
acc(2)
```

检查值的类型

TYPE 语句用于告诉值的类型，或者使用预定义的类型字符串生成类型信息。例如：

```
PRINT TYPE(123); TYPE("Hi"); ' 获取值的类型
PRINT TYPE("INT"); TYPE("REAL"); ' 获取特定类型
```

还可以使用 IS 运算符来检查一个值是否与特定类型匹配，如下所示：

```
PRINT 123 IS TYPE("INT"); "Hi" IS TYPE("STRING");
PRINT inst IS TYPE("CLASS");
```

IS 语句还可以判断一个原型的实例是否继承自另一个原型：

```
PRINT inst IS Foo; ' 如果 foo 是 inst 的原型，则为真
```

将类型值传递给 STR 语句，以获取类型名称的字符串。

导入另一个BASIC文件

在大型项目中，通过将不同部分分离到多个文件中，有助于组织源代码。IMPORT 语句导入其他源文件，就好像它们是在导入的位置编写的一样。例如，假设我们有一个名为 "a.bas" 的文件：

```
foo = 1
```

还有另一个名为 "b.bas" 的文件：

```
IMPORT "a.bas"
PRINT Foo;
```

现在，您可以使用从 "a.bas" 导入的所有内容。MY-BASIC 可以正确处理循环导入。

导入模块
还可以将一些本地脚本接口放在一个模块（也称为“命名空间”）中，以避免命名污染。目前 MY-BASIC 不支持在 BASIC 中创建模块。使用 IMPORT "@xxx" 来导入一个本地模块，该模块中的所有符号都可以无需模块前缀而使用。

3.核心和标准库

MY-BASIC 提供了一组常用函数，这些函数提供了基本的数字和字符串操作。这些函数名不能用作用户自定义的标识符。有关这些函数的详细信息，请参见下面的图表：

Type	Name	Description
Numeric	ABS	返回一个数字的绝对值。
	SGN	返回一个数字的符号。
	SQR	返回一个数字的算术平方根。
	FLOOR	返回不大于一个数字的最大整数。
	CEIL	返回不小于一个数字的最小整数。
	FIX	返回一个数字的整数部分。
	ROUND	返回一个数字的最接近的近似整数。
	SRND	种子随机数生成器。
	RND	返回一个介于 [0.0, 1.0] 之间的随机浮点数，或者介于 [0, max] 之间的随机整数（通过 RND(max)），或者介于 [MIN, MAX] 之间的随机数（通过 RND(min, max)）。
	SIN	返回一个数字的正弦值。
	COS	返回一个数字的余弦值。
	TAN	返回一个数字的正切值。
	ASIN	返回一个数字的反正弦值。
	ACOS	返回一个数字的反余弦值。
	ATAN	返回一个数字的反正切值。
	EXP	返回一个数字的以 e 为底的指数函数值。
	LOG	返回一个数字的以 e 为底的对数值。

String	ASC	返回一个字符的 ASCII 码整数值。
	CHR	返回一个 ASCII 码的字符。
	LEFT	返回字符串左侧指定数量的字符。
	MID	返回字符串从指定位置开始的特定数量字符。
	RIGHT	返回字符串右侧指定数量的字符。
	STR	返回一个数字的字符串表示，或使用 TO_STRING 函数格式化一个类实例。
Common	VAL	返回字符串的数值表示，或字典迭代器的值，可重写用于引用的用户类型和类实例。
	LEN	返回字符串或数组的长度，或 LIST 或 DICT 的元素计数，可重写用于引用的用户类型和类实例。
Input & Output	PRINT	将数字或字符串输出到标准输出流，可由用户重定向。
	INPUT	从标准输入流输入数字或字符串，可由用户重定向。

INPUT 语句后面可以跟一个可选的输入提示字符串，然后是要填充的变量标识符；当指定 $ 时，它接受字符串，否则接受数字。注意，除了 PRINT 和 INPUT 之外的所有这些函数都需要用一对括号括住参数；RND 语句有点特殊，它可以带括号，也可以不带括号，详细情况请参见图表。

4.集合库

MY-BASIC 提供了一组 LIST 和 DICT 操作函数，提供了创建、访问、迭代等功能，如下所示：

Name	Description
LIST	创建列表
DICT	创建字典
PUSH	将一个值推送到列表的尾部，可覆盖引用的用户类型和类实例
POP	从列表尾部弹出一个值，可覆盖引用的用户类型和类实例
BACK	查看列表尾部的值，可覆盖引用的用户类型和类实例
INSERT	在列表的特定位置插入一个值，对于引用的用户类型和类可重写实例
SORT	对列表进行递增排序，可覆盖以供引用用户类型和类实例
EXISTS	告诉列表是否包含特定值，或者字典是否包含特定键，可覆盖引用的用户类型和类实例
INDEX_OF	获取列表中值的索引，对于引用的用户类型和类实例可覆盖
GET	返回列表中特定索引处的值，或字典中具有特定键的值，或类实例的成员，可覆盖引用的用户类型和类实例
SET	设置列表中特定索引处的值，或字典中具有特定键的值，或类实例的成员变量，对于引用的用户类型和类实例可覆盖

REMOVE	删除列表中特定索引处的元素，或字典中具有特定键的元素，对于引用的用户类型和类实例可覆盖
CLEAR	清除列表或字典，可覆盖引用的用户类型和类实例
CLONE	克隆集合或引用的用户类型
TO_ARRAY	将列表中的所有元素复制到数组
ITERATOR	获取列表或字典的迭代器，对于引用的用户类型和类可重写实例
MOVE_NEXT	将迭代器移动到列表或字典上的下一个位置，对于引用的用户类型和类实例可覆盖

使用集合，例如:

```
l = list(1, 2, 3, 4)
set(l, 1, "B")
print exists(l, 2); pop(l); back(l); len(l);
    d = dict(1, "One", 2, "Two")
    set(d, 3, "Three")
print len(d);
it = iterator(d)
while move_next(it)
print get(it);
wend
```

MY-BASIC 支持直接使用括号访问列表或字典中的元素：

```
d = dict()
d(1) = 2
print d(1);
```

列表从零开始，数组在 MY-BASIC 中也是如此。

5.应用程序接口

MY-BASIC 是用标准 C 语言在双文件中干净地编写的。 将 MY-BASIC 嵌入现有项目所需要做的就是将 my_basic.h 和 my_basic.c 复制到目标项目，然后将它们添加到项目构建管道中。 所有接口均在 my_basic.h 中声明。 大多数API返回代表执行状态的int值，如果没有错误，大多数应该返回 MB_FUNC_OK，详细信息请检查my_basic.h中的MB_CODES宏。 但也有一些例外情况，即简单的 int 不适合。

解释器结构
MY-BASIC 使用解释器结构来在解析和运行过程中存储必要的数据，包括注册的函数、抽象语法树（AST）、解析上下文、运行上下文、作用域、错误信息等。解释器结构是上下文的一个单位。

元信息

```
unsigned long mb_ver(void);
```

返回当前解释器的版本号。

```
const char* mb_ver_string(void);
```

返回当前解释器的版本文本。

初始化和释放

```
int mb_init(void);
```

在执行任何其他操作之前，必须且仅能调用一次此函数，以初始化整个系统。

```
int mb_dispose(void);
```

在使用 MY-BASIC 后，必须且仅能调用一次此函数，以释放整个系统。

```
int mb_open(struct mb_interpreter_t** s);
```

此函数打开一个解释器实例，准备进行解析和运行。
通常使用方法如下：

```
struct mb_interpreter_t* bas = 0;
mb_open(&bas);
```

```
int mb_close(struct mb_interpreter_t** s);
```

此函数在使用后关闭一个解释器实例。mb_open 和 mb_close 必须成对匹配。

```
int mb_reset(struct mb_interpreter_t** s, bool_t clear_funcs, bool_t clear_vars);
```

此函数重置解释器实例，通常用于新的加载-运行循环。如果 clear_vars 为 true，则清除所有变量；如果 clear_funcs 为 true，则清除所有已注册的全局函数，但此函数不会重新注册内置接口。

分叉
这些函数用于分叉和加入解释器。

```
int mb_fork(struct mb_interpreter_t** s, struct mb_interpreter_t* r, bool_t
clear_forked);
```

此函数从 r 分叉出一个新的解释器，保存到 s。所有分叉出的环境共享相同的注册函数、解析的代码等，但使用自己的运行上下文。将 clear_forked 设置为 true，以便让源实例在分叉的实例中收集和管理数据。

```
int mb_join(struct mb_interpreter_t** s);
```

此函数加入一个已分叉的解释器。使用此函数关闭已分叉的解释器。

```
int mb_get_forked_from(struct mb_interpreter_t* s, struct mb_interpreter_t** src);
```

此函数获取一个已分叉解释器的源解释器。
函数注册/注销
这些函数用于注册或注销本地脚本接口。

```
int mb_register_func(struct mb_interpreter_t* s, const char* n, mb_func_t f);
```

此函数将函数指针注册到一个解释器中，指定名称。函数必须具有以下签名：int (* mb_func_t)(struct mb_interpreter_t*, void**)。已注册的函数可以在 MY-BASIC 代码中调用。此函数返回受影响的条目数量，因此非零表示成功。指定的标识符将以大写字母存储。

与核心交互
以下这些函数用于在扩展函数中与核心进行通信。

```
int mb_attempt_func_begin(struct mb_interpreter_t* s, void** l);
```

此函数检查 BASIC 是否以合法的方式调用扩展函数的起始。在不带参数的情况下开始扩展函数时调用它。

```
int mb_attempt_func_end(struct mb_interpreter_t* s, void** l);
```

此函数检查 BASIC 是否以合法的方式结束扩展函数的调用。在不带参数的情况下结束扩展函数时调用它。

```
int mb_attempt_open_bracket(struct mb_interpreter_t* s, void** l);
```

此函数检查 BASIC 是否以合法的方式调用扩展函数，以开放括号开始。

```
int mb_attempt_close_bracket(struct mb_interpreter_t* s, void** l);
```

此函数检查 BASIC 是否以合法的方式调用扩展函数，在参数列表后以闭合括号结束。

```
int mb_has_arg(struct mb_interpreter_t* s, void** l);
```

此函数检测当前执行位置是否还有更多的参数。使用此函数来实现可变参数函数。如果没有更多参数，则返回零，否则返回非零值。

```
int mb_pop_int(struct mb_interpreter_t* s, void** l, int_t* val);
```

此函数尝试从解释器中弹出一个 int_t 参数，并将结果存储在 *val 中。

```
int mb_pop_real(struct mb_interpreter_t* s, void** l, real_t* val);
```

此函数尝试从解释器中弹出一个 real_t 参数，并将结果存储在 *val 中。

```
int mb_pop_string(struct mb_interpreter_t* s, void** l, char** val);
```

此函数尝试从解释器中弹出一个 char*（字符串）参数，并将指针存储在 *val 中。您不需要知道何时弹出的字符串将被释放，但请注意，弹出的字符串可能在弹出下一个字符串参数时被释放，因此请及时处理或缓存它。

```
int mb_pop_usertype(struct mb_interpreter_t* s, void** l, void** val);
```

此函数尝试从解释器中弹出一个 void*（用户类型）参数，并将结果存储在 *val 中。如果用户类型大于 void，请改用 mb_pop_value。

```
int mb_pop_value(struct mb_interpreter_t* s, void** l, mb_value_t* val);
```

此函数尝试从解释器中弹出一个 mb_value_t 参数，并将结果存储在 *val 中。如果扩展函数接受不同类型的参数，或者弹出其他高级数据类型，则使用此函数代替 mb_pop_int、mb_pop_real 和 mb_pop_string。

```
int mb_push_int(struct mb_interpreter_t* s, void** l, int_t val);
```

此函数将一个 int_t 值推送到解释器中。

```
int mb_push_real(struct mb_interpreter_t* s, void** l, real_t val);
```

此函数将一个 real_t 值推送到解释器中。

```
int mb_push_string(struct mb_interpreter_t* s, void** l, char* val);
```

此函数将一个 char *（字符串）值推送到解释器中。char* val 的内存必须由 MY-BASIC 分配并释放。在推送之前使用 mb_memdup 进行分配。例如：

```
mb_push_string(s, l, mb_memdup(str, (unsigned)(strlen(str) + 1)));
```

```
int mb_push_usertype(struct mb_interpreter_t* s, void** l, void* val);
```

此函数将一个 void *（用户类型）值推送到解释器中。如果用户类型大于 void，请改用 mb_push_value。

```
int mb_push_value(struct mb_interpreter_t* s, void** l, mb_value_t val);
```

此函数将一个 mb_value_t 值推送到解释器中。如果扩展函数返回通用类型，或者推送其他高级数据类型，则使用此函数代替 mb_push_int、mb_push_real 和 mb_push_string。

类定义
这些函数用于在本机端手动定义一个类。

```
int mb_begin_class(struct mb_interpreter_t* s, void** l, const char* n, mb_value_t**
meta, int c, mb_value_t* out);
```

此函数以特定名称开始一个类定义。n 是大写的类名。meta 是 mb_value_t* 的数组，它们是可选的元类；c 是元数组的元素计数。生成的类将返回到 *out 中。

```
int mb_end_class(struct mb_interpreter_t* s, void** l);
```

此函数结束类定义。

```
int mb_get_class_userdata(struct mb_interpreter_t* s, void** l, void** d);
```

此函数获取类实例的用户数据。返回的数据将存储到 *d 中。

```
int mb_set_class_userdata(struct mb_interpreter_t* s, void** l, void* d);
```

此函数使用数据 d 设置类实例的用户数据。

值操作
这些函数用于操作值。

```
int mb_get_value_by_name(struct mb_interpreter_t* s, void** l, const char* n,
mb_value_t* val);
```

此函数通过指定的名称（大写）获取一个标识符的值。n 是预期的名称文本。它将返回一个值到 *val。

```
int mb_get_vars(struct mb_interpreter_t* s, void** l, mb_var_retrieving_func_t r, int
stack_offset);
```

此函数检索特定堆栈帧中活动变量的名称和值，然后返回检索的变量计数。stack_offset 表示堆栈偏移量，0 表示当前帧，-1 表示根，其他正值表示从顶部（活动）帧的偏移量。

```
int mb_add_var(struct mb_interpreter_t* s, void** l, const char* n, mb_value_t val,
bool_t force);
```

此函数向解释器中添加一个具有特定标识符名称（大写）和值的变量。n 是名称文本，val 是变量的值，force 表示是否覆盖现有值。

```
int mb_get_var(struct mb_interpreter_t* s, void** l, void** v, bool_t redir);
```

此函数获取一个标记，如果它是变量，则将其存储在参数 *v 中。redir 表示是否将结果变量重定向到类实例的任何成员变量。

```
int mb_get_var_name(struct mb_interpreter_t* s, void** v, char** n);
```

此函数获取变量的名称，然后将其存储在参数 *n 中。

```
int mb_get_var_value(struct mb_interpreter_t* s, void** l, mb_value_t* val);
```

此函数将变量的值获取到 *val 中。

```
int mb_set_var_value(struct mb_interpreter_t* s, void** l, mb_value_t val);
```

此函数将值从 val 设置为变量的值。

```
int mb_init_array(struct mb_interpreter_t* s, void** l, mb_data_e t, int* d, int c,
void** a);
```

此函数初始化可以在 BASIC 中使用的数组。参数 mb_data_e t 代表数组中元素的类型，可以是 MB_DT_REAL 或 MB_DT_STRING；禁用 MB_SIMPLE_ARRAY 宏以使用复杂数组，将 MB_DT_NIL 传递给它。int* d 和 int c 代表维度的等级和维度计数。函数将创建的数组存储在 void** a 中。

```
int mb_get_array_len(struct mb_interpreter_t* s, void** l, void* a, int r, int* i);
```

此函数获取数组的长度。int r 表示要获取的维度。

```
int mb_get_array_elem(struct mb_interpreter_t* s, void** l, void* a, int* d, int c,
mb_value_t* val);
```

此函数获取数组中一个元素的值。

```
int mb_set_array_elem(struct mb_interpreter_t* s, void** l, void* a, int* d, int c,
mb_value_t val);
```

此函数设置数组中一个元素的值。

```
int mb_init_coll(struct mb_interpreter_t* s, void** l, mb_value_t* coll);
```

此函数初始化一个集合；传递一个有效的 mb_value_t 指针，其中包含您希望初始化的特定集合类型。

```
int mb_get_coll(struct mb_interpreter_t* s, void** l, mb_value_t coll, mb_value_t
idx, mb_value_t* val);
```

此函数获取集合中的一个元素。它接受 LIST 索引或 DICT 键，使用 mb_value_t idx。

```
int mb_set_coll(struct mb_interpreter_t* s, void** l, mb_value_t coll, mb_value_t
idx, mb_value_t val);
```

此函数设置集合中的一个元素。它接受 LIST 索引或 DICT 键，使用 mb_value_t idx。

```
int mb_remove_coll(struct mb_interpreter_t* s, void** l, mb_value_t coll, mb_value_t
idx);
```

此函数从集合

中删除一个元素。它接受 LIST 索引或 DICT 键，使用 mb_value_t idx。

```
int mb_count_coll(struct mb_interpreter_t* s, void** l, mb_value_t coll, int* c);
```

此函数返回集合中的元素数量。

```
int mb_keys_of_coll(struct mb_interpreter_t* s, void** l, mb_value_t coll,
mb_value_t* keys, int c);
```

此函数检索集合的所有键。它获取 LIST 的索引或 DICT 的键，并将它们存储在 mb_value_t* keys 中。

```
int mb_make_ref_value(struct mb_interpreter_t* s, void* val, mb_value_t* out,
mb_dtor_func_t un, mb_clone_func_t cl, mb_hash_func_t hs, mb_cmp_func_t cp,
mb_fmt_func_t ft);
```

此函数创建一个包含 void* val 作为原始用户数据的引用 usertype mb_value_t 对象。注意，您需要为核心提供一些函数来操纵它。

```
int mb_get_ref_value(struct mb_interpreter_t* s, void** l, mb_value_t val, void**
out);
```

此函数从引用 usertype 获取原始用户数据。

```
int mb_ref_value(struct mb_interpreter_t* s, void** l, mb_value_t val);
```

此函数增加引用值的引用计数。

```
int mb_unref_value(struct mb_interpreter_t* s, void** l, mb_value_t val);
```

此函数减少引用值的引用计数。

```
int mb_set_alive_checker(struct mb_interpreter_t* s, mb_alive_checker_t f);
```

此函数全局设置对象存活性检查器。

```
int mb_set_alive_checker_of_value(struct mb_interpreter_t* s, void** l, mb_value_t
val, mb_alive_value_checker_t f);
```

此函数在特定引用 usertype 值上设置对象存活性检查器。

```
int mb_override_value(struct mb_interpreter_t* s, void** l, mb_value_t val,
mb_meta_func_e m, void* f);
```

此函数重写特定引用 usertype 值的元函数。

```
int mb_dispose_value(struct mb_interpreter_t* s, mb_value_t val);
```

此函数用于释放从解释器弹出的值。仅用于字符串。
调用操作

```
int mb_get_routine(struct mb_interpreter_t* s, void** l, const char* n, mb_value_t*
val);
```

此函数通过名称（大写）获取一个可调用值。

```
int mb_set_routine(struct mb_interpreter_t* s, void** l, const char* n,
mb_routine_func_t f, bool_t force);
```

此函数使用本机函数设置特定名称（大写）的可调用值。

```
int mb_eval_routine(struct mb_interpreter_t* s, void** l, mb_value_t val, mb_value_t*
args, unsigned argc, mb_value_t* ret);
```

此函数评估一个可调用值。mb_value_t* args 是指向参数数组的指针，unsigned argc 是其数量。最后一个可选参数 mb_value_t* ret 接收返回值；如果未使用，则传递 NULL。

```
int mb_get_routine_type(struct mb_interpreter_t* s, mb_value_t val,
mb_routine_type_e* y);
```

此函数获取可调用值的子类型。mb_value_t val 是可调用值的值，结果将分配给 mb_routine_type_e* y。

解析和运行

```
int mb_load_string(struct mb_interpreter_t* s, const char* l, bool_t reset);
```

此函数将字符串加载到解释器中，然后将 BASIC 源代码解析为可执行结构，并将其附加到 AST。

```
int mb_load_file(struct mb_interpreter_t* s, const char* f);
```

此函数将文件加载到解释器中，然后将 BASIC 源代码解析为可执行结构，并将其附加到 AST。

```
int mb_run(struct mb_interpreter_t* s, bool_t clear_parser);
```

此函数在解释器中运行解析的 AST。

```
int mb_suspend(struct mb_interpreter_t* s, void** l);
```

（已过时。）此函数暂停并保存当前执行点。再次调用 mb_run 可以从暂停点恢复。

```
int mb_schedule_suspend(struct mb_interpreter_t* s, int t);
```

（已过时。）此函数安排一个挂起事件，它将在完成活动语句后触发该事件。在整个执行过程中需要执行其他操作时，这很有用。
A)。 mb_schedule_suspend(s, MB_FUNC_SUSPEND); 它是可重新进入的，这意味着下一次 mb_run 将从暂停的地方继续执行。B)。 mb_schedule_suspend(s, MB_FUNC_END); 正常终止执行，无错误消息。C)。 mb_schedule_suspend(s, MB_EXTENDED_ABORT); 或者传递大于 MB_EXTENDED_ABORT 的参数，以终止执行并触发错误消息。您可以在 _on_stepped 中或在脚本接口函数中调用 mb_schedule_suspend。mb_schedule_suspend 与 mb_suspend 的区别在于 mb_suspend 只能在脚本接口中调用，不能捕获 B) 和 C) 类型的挂起。

调试

```
int mb_debug_get(struct mb_interpreter_t* s, const char* n, mb_value_t* val);
```

此函数检索具有特定名称（大写）的变量的值。

```
int mb_debug_set(struct mb_interpreter_t* s, const char* n, mb_value_t val);
```

此函数设置具有特定名称（大写）的变量的值。

```
int mb_debug_count_stack_frames(struct mb_interpreter_t* s);
```

此函数获取并返回堆栈帧数。

```
int mb_debug_get_stack_trace(struct mb_interpreter_t* s, char** fs, unsigned fc);
```

此函数跟踪当前调用堆栈。要使用此函数，需要启用 MB_ENABLE_STACK_TRACE 宏。

```
int mb_debug_set_stepped_handler(struct mb_interpreter_t* s, mb_debug_stepped prev,
mb_debug_stepped post);
```

此函数设置解释器的一对步骤处理程序。要设置的函数必须是 int (*mb_debug_stepped_handler_t)(struct mb_interpreter_t*, void*, const char*, int, unsigned short, unsigned short) 类型的指针。这些函数对于逐步调试或在执行过程中处理额外的内容非常有用，prev 和 post 处理程序分别在语句执行之前和之后调用。

类型处理

```
const char* mb_get_type_string(mb_data_e t);
```

此函数返回特定类型的字符串值。

错误处理

```
int mb_raise_error(struct mb_interpreter_t* s, void** l, mb_error_e err, int ret);
```

此函数手动引发错误。

```
mb_error_e mb_get_last_error(struct mb_interpreter_t* s, const char** file, int* pos,
unsigned short* row, unsigned short* col);
```

此函数在解释器结构中返回最新的错误信息和详细位置。它还清除最新的错误信息。

```
const char* mb_get_error_desc(mb_error_e err);
```

此函数返回特定错误的字符串值。

```
int mb_set_error_handler(struct mb_interpreter_t* s, mb_error_handler_t h);
```

此函数设置解释器的错误处理程序。

IO 重定向

```
int mb_set_printer(struct mb_interpreter_t* s, mb_print_func_t p);
```

此函数设置解释器的 PRINT 处理程序。使用此函数为 PRINT 语句自定义输出处理程序。要设置的函数必须是 int (*mb_print_func_t)(struct mb_interpreter_t*, const char*, ...) 类型的指针。默认为 printf。

```
int mb_set_inputer(struct mb_interpreter_t* s, mb_input_func_t p);
```

此函数设置解释器的 INPUT 处理程序。使用此函数为 INPUT 语句自定义输入处理程序。要设置的函数必须是 int (*mb_input_func_t)(struct mb_interpreter_t*, const char*, char*, int) 类型的指针。默认为 mb_gets。第一个参数是可选的提示文本，例如 `INPUT "Some text", A$`。如果不需要提示文本，可以省略它。

杂项

```
int mb_set_import_handler(struct mb_interpreter_t* s, mb_import_handler_t h);
```

此函数为 BASIC 代码设置自定义的导入处理程序。

```
int mb_set_memory_manager(mb_memory_allocate_func_t a, mb_memory_free_func_t f);
```

此函数全局设置 MY-BASIC 的内存分配器和释放器。

```
bool_t mb_get_gc_enabled(struct mb_interpreter_t* s);
```

此函数获取垃圾回收是否启用。

```
int mb_set_gc_enabled(struct mb_interpreter_t* s, bool_t gc);
```

此函数设置垃圾回收是否启用，可用于暂停和恢复垃圾回收。

```
int mb_gc(struct mb_interpreter_t* s, int_t* collected);
```

此函数尝试手动触发垃圾回收，并获取已回收的内存量。

```
int mb_get_userdata(struct mb_interpreter_t* s, void** d);
```

此函数获取解释器实例的用户数据。

```
int mb_set_userdata(struct mb_interpreter_t* s, void* d);
```

此函数设置解释器实例的用户数据。

```
int mb_gets(struct mb_interpreter_t* s, const char* pmt, char* buf, int s);
```

此函数是标准 C gets 的更安全版本。返回输入文本的长度。

```
char* mb_memdup(const char* val, unsigned size);
```

此函数复制一个 MY-BASIC 可管理的缓冲区内存块。在推送字符串参数之前使用此函数。注意，此函数仅分配并复制指定大小的字节，因此必须将大小增加一个额外的字节用于结束符“\0”。例如：

```
mb_push_string(s, l, mb_memdup(str, (unsigned)(strlen(str) + 1)));
```

6.使用 MY-BASIC 进行脚本编程

C 编程语言在源代码可移植性方面表现出色，因为几乎每个平台都有 C 编译器可用。MY-BASIC 使用标准 C 编写，因此可以在不同的平台上进行编译，几乎不需要或只需要少量修改。将 MY-BASIC 嵌入到现有项目中也非常简单，只需添加核心部分的两个文件即可。

应该认识到，在您的项目中，哪些部分对执行速度敏感，哪些部分对灵活性和可配置性敏感。脚本适用于易变的部分。

7.定制 MY-BASIC

重定向 PRINT 和 INPUT
包含一个标头文件以使用变参：

```
#include <stdarg.h>
```

自定义一个打印处理程序，例如：

```
int my_print(const char* fmt, ...) {
    char buf[128];
    char* ptr = buf;
    size_t len = sizeof(buf);
    int result = 0;
    va_list argptr;
    va_start(argptr, fmt);
    result = vsnprintf(ptr, len, fmt, argptr);
    if (result < 0) {
        fprintf(stderr, "Encoding error.\n");
    } else if (result > (int)len) {
        len = result + 1;
        ptr = (char*)malloc(result + 1);
        result = vsnprintf(ptr, len, fmt, argptr);
    }
    va_end(argptr);
    if (result ≥ 0)
        printf(ptr); /* 在此更改 */
    if (ptr ≠ buf)
        free(ptr);
    return result;
}
```

自定义一个输入处理程序，例如：

```
int my_input(const char* pmt, char* buf, int s) {
    int result = 0;
    if (fgets(buf, s, stdin) == 0) { /* 在此更改 */
        fprintf(stderr, "Error reading.\n");
        exit(1);
    }
    result = (int)strlen(buf);
    if (buf[result - 1] == '\n')
        buf[result - 1] = '\0';
    return result;
}
```

将这些处理程序注册到解释器：

```
mb_set_printer(bas, my_print);
mb_set_inputer(bas, my_input);
```

现在，您的自定义打印处理程序和输入处理程序将代替标准处理程序。在 BASIC 中使用 PRINT 和 INPUT 来访问它们。

编写脚本API
您也可以引入自己的脚本接口。
首先，在本地编程语言（通常是C）中定义函数。由BASIC调用的扩展函数是一个 int (*)(*struct mb_interpreter_t*, void**) 类型的指针。解释器实例作为扩展函数的第一个参数，函数可以从解释器结构中弹出变参，并将零个或一个返回值推回到结构中。int 返回值表示扩展函数的执行状态，无错误应始终返回 MB_FUNC_OK。让我们以编写一个返回两个整数中的最大值的函数为例进行教程；请参见以下代码：

```
int maximum(struct mb_interpreter_t* s, void** l) {
    int result = MB_FUNC_OK;
    int m = 0;
    int n = 0;
    int r = 0;
    mb_assert(s && l);
    mb_check(mb_attempt_open_bracket(s, l));
    mb_check(mb_pop_int(s, l, &m));
    mb_check(mb_pop_int(s, l, &n));
    mb_check(mb_attempt_close_bracket(s, l));
    r = m > n ? m : n;
    mb_check(mb_push_int(s, l, r));
    return result;
}
```

第二步是通过 mb_reg_fun(bas, maximum) 注册这个函数（假设我们已经初始化了 struct mb_interpreter_t* bas）。
然后，您可以在MY-BASIC中像使用其他BASIC函数一样使用它，例如：

```
i = maximum(1, 2)
print i;
```

只需返回大于或等于宏 MB_EXTENDED_ABORT 的整数值来表示用户定义的中止。建议添加一个中止值，例如：

```
typedef enum mb_user_abort_e {
    MB_ABORT_FOO = MB_EXTENDED_ABORT + 1,
    /* 更多中止枚举 ... */
};
```

然后，在扩展函数中出现错误时，使用`return MB_ABORT_FOO;`。

使用用户类型值
当内置类型不适用时，可以考虑使用用户类型（usertypes）。它可以接受您提供的任何数据。MY-BASIC不关心用户类型的实际内容；它只持有一个用户类型值，并与BASIC进行通信。
获取或设置用户类型的接口只有两个：mb_pop_usertype 和 mb_push_usertype。您可以将 void* 推入解释器，然后将 void* 作为值弹出。

有关使用引用用户类型的更多信息，请参考上面的接口，或查看网站。

宏
MY-BASIC 的一些特性可以通过宏进行配置。

MB_SIMPLE_ARRAY
默认启用。整个数组使用统一的类型标记，这意味着只有两种类型的数组，字符串和实数类型。
如果希望在数组中存储通用类型的值，包括整数类型（int_t）、实数类型（real_t）、用户类型等，可以禁用此宏。此外，字符串数组仍然是另一种类型。请注意，非简单数组需要额外的内存来存储每个元素的类型标记。

MB_ENABLE_ARRAY_REF
默认启用。如果定义了此宏，编译为引用数组；否则，编译为值类型数组。

MB_MAX_DIMENSION_COUNT
默认为 4。将其更改为支持更多的数组维度。请注意，它不能大于无符号 char 可以持有的最大数字。

MB_ENABLE_COLLECTION_LIB
默认启用。如果定义了此宏，编译时将包括 LIST 和 DICT 库。

MB_ENABLE_USERTYPE_REF
默认启用。如果定义了此宏，编译时将包括引用用户类型支持。

MB_ENABLE_ALIVE_CHECKING_ON_USERTYPE_REF
默认启用。如果定义了此宏，编译时将包括引用用户类型的对象存活性检查。

MB_ENABLE_CLASS
默认启用。如果定义了此宏，编译时将包括类（原型）支持。

MB_ENABLE_LAMBDA
默认启用。如果定义了此宏，编译时将包括 lambda（匿名函数）支持。

MB_ENABLE_MODULE
默认启用。如果定义了此宏，编译时将包括模块（命名空间）支持。使用 IMPORT “@xxx” 来导入一个模块，可以在不带模块前缀的情况下使用该模块中的所有符号。

MB_ENABLE_UNICODE
默认启用。如果定义了此宏，编译时将包括 UTF8 处理功能，以便使用诸如 LEN、LEFT、RIGHT、MID 等函数正确处理 UTF8 字符串。

MB_ENABLE_UNICODE_ID
默认启用。如果定义了此宏，编译时将包括 UTF8 令牌支持。此功能需要启用 MB_ENABLE_UNICODE。

MB_ENABLE_FORK
默认启用。如果定义了此宏，编译时将包括 fork 支持。

MB_GC_GARBAGE_THRESHOLD
默认为 16。当发生此数量的释放时，将触发一次扫描-回收 GC 循环。

MB_ENABLE_ALLOC_STAT
默认启用。使用 MEM 命令告知 MY-BASIC 已分配了多少字节的内存。请注意，每次分配的统计信息需要多占用 sizeof(intptr_t) 字节。

MB_ENABLE_SOURCE_TRACE
默认启用。跟踪错误发生的位置。如果在内存敏感的平台上，可以禁用此功能以减少一些内存占用。

MB_ENABLE_STACK_TRACE
默认启用。如果定义了此宏，MY-BASIC 将记录包括子例程和本地函数在内的堆栈帧。

MB_ENABLE_FULL_ERROR
默认启用。显示详细的错误消息。否则，所有错误类型将显示“发生错误”消息。但是，通过检查回调中的错误代码，始终可以获取特定的错误类型。

MB_CONVERT_TO_INT_LEVEL
描述在表达式评估后如何处理实数。如果定义了 MB_CONVERT_TO_INT_LEVEL_NONE，则将其保留为实数；否则，在定义为 MB_CONVERT_TO_INT_LEVEL_ALL 时，尝试将其转换为整数，如果不包含小数部分的话。您还可以在自己的脚本接口函数中使用 mb_convert_to_int_if_posible 宏来处理 mb_value_t。

MB_PRINT_INPUT_PROMPT
默认启用。更倾向于将指定的输入提示输出到解释器的输出函数。

MB_PRINT_INPUT_CONTENT
默认禁用。更倾向于将输入内容输出到解释器的输出函数。

MB_PREFER_SPEED
默认启用。尽可能优

先考虑运行速度，而不是空间占用。禁用此选项可以减少内存占用。

MB_COMPACT_MODE
默认启用。C 结构可能使用紧凑布局。这可能会导致某些编译器（例如某些嵌入式系统）出现一些奇怪的指针访问错误。如果遇到任何奇怪的错误，请尝试禁用此选项。

_WARNING_AS_ERROR
默认为 0。在 my_basic.c 中将此宏定义为 1，以将警告视为错误，否则它们将被静默忽略。例如，除以零、传递错误类型的参数将触发警告。

_HT_ARRAY_SIZE_DEFAULT
默认为 193。在 my_basic.c 中更改此值以调整哈希表的大小。较小的值将减少一些内存占用，哈希表的大小将影响加载期间的标记化和解析时间，通常不会对运行性能产生影响（除了跨作用域标识符查找）。

_SINGLE_SYMBOL_MAX_LENGTH
默认为 128。词法符号的最大长度。

8.内存占用

在某些内存受限的平台上，内存占用通常是一个敏感的瓶颈。MY-BASIC 提供了一种方法来统计解释器分配了多少内存。编写以下脚本以以字节为单位显示内存使用量：

```
print mem;
```

请注意，如果启用了此统计信息，每次分配将额外占用 sizeof(intptr_t) 字节，但这些额外的字节不计入内存使用量。

通过删除 my_basic.h 中的 MB_ENABLE_SOURCE_TRACE 宏来禁用源代码跟踪，以减少一些内存占用，但仍然会显示错误。

在 my_basic.c 中重新定义 _HT_ARRAY_SIZE_DEFAULT 宏，将其最小值设置为 1，以减少 MY-BASIC 中哈希表占用的内存。值 1 表示线性查找，主要用于解析机制和复杂标识符的动态查找。

在运行时间很长、内存有限的嵌入式系统中，由于碎片化，内存可能被严重浪费。此外，即使在内存充足的系统上，为 MY-BASIC 定制内存分配器也是有效的。

内存分配器的形式如下：

```
typedef char* (* mb_memory_allocate_func_t)(unsigned s);
```

释放器的形式如下：

```
typedef void (* mb_memory_free_func_t)(char* p);
```

然后，您可以通过以下方式全局地指示 MY-BASIC 使用它们，而不是标准的 malloc 和 free 函数：

```
MBAPI int mb_set_memory_manager(mb_memory_allocate_func_t a, mb_memory_free_func_t
f);
```

请注意，这些函数只会影响 my_basic.c 内部的事物，而 main.c 仍然使用标准的 C 库。

在 main.c 中已经有一个简单的内存池实现。您需要确保 _USE_MEM_POOL 宏被定义为 1 以启用它。

在此实现中有四个函数可作为示范：*open_mem_pool 在设置解释器时打开内存池；*close_mem_pool 在终止时关闭内存池；一对 *pop_mem 和 _push_mem 被注册到 MY-BASIC 中。请注意，如果预期的大小不是内存池中的常见大小，*pop_mem 将调用标准的 malloc 函数；并且每次分配将额外占用 sizeof(union _pool_tag_t) 字节来存储元数据。典型的工作流程如下所示：

```
_open_mem_pool(); // 打开内存池
mb_set_memory_manager(_pop_mem, _push_mem); // 注册它们
{
    mb_init();
    mb_open(&bas);
    // 处理 MY-BASIC
    mb_close(&bas);
    mb_dispose();
}
_close_mem_pool(); // 完成
```

严格来说，内存池不能保证在连续空间分配内存，它是一个对象池而不是内存池，它将一个具有预期大小的自由内存块弹出给用户，当用户释放内存块时，将其推回堆栈，而不是释放到系统。您可以将其替换为其他高效的算法，以获得良好的性能和在空间和速度之间的平衡。

9.将 MY-BASIC 用作独立解释器

直接执行二进制文件，不需要任何参数，即可启动交互模式。以下是此模式下的命令：

Command	Summary	Usage
HELP	检视帮助信息	
CLS	清屏	
NEW	清除当前程序	
RUN	运行当前程序	
BYE	退出解释器	
LIST	列出当前程序	LIST [l [n]]，其中 l 为起始行号，n 为行数。
EDIT	编辑（修改/插入/删除）当前程序中的一行	EDIT n，其中 n 为行号； EDIT -i n，在特定行之前插入一行，n 为行号； EDIT -r n，删除一行，n 为行号。

LOAD	将文件加载到当前程序中。	LOAD .
SAVE	将当前程序保存到文件中。	SAVE .
KILL	删除一个文件	KILL .
DIR	列出目录中的所有文件	DIR p，其中 p 为目录路径。

输入一个命令（可能带有多个必要的参数），然后提示按回车键执行它。命令只涉及解释器的操作，而不是关键字。

将文件路径传递给二进制文件，以立即加载并运行该BASIC文件。

传递选项 -e 和一个表达式以立即求值并打印它，例如 -e "22 / 7"，请注意，当表达式包含空格字符时，需要使用双引号。

传递选项 -p 和一个数字以设置内存池的阈值大小，例如 -p 33554432 以将阈值设置为32MB。当空闲列表达到此大小时，MY-BASIC将整理内存池。