Version 13, 1 May 2005

This document describes the assumptions made by the Pyth run-time about the representation of objects, values of variables, functions, virtual tables, and stack frames. User code generated by the compiler must conform to these assumptions, on pain of extremely obscure errors.

1 Restrictions in Pyth

Pyth is a dialect of Python, and does not support all of Python's semantics. Here are the most relevant changes, some of which have changed since Project #2.

1. Pyth has a different set of built-in types, methods, functions, and global variables. You needn't generally worry about these specifically, since you get them automatically in the standard prelude.

2. Where Python uses a special boolean type, Pyth uses Int (integer), and the values True and False are simply variables defined in the standard prelude.

3. Classes and types are not valid as values. The assignment

   ```
   a = Int
   ```

   is not valid. This is enforced in the front end, and your code generator can assume that no program will attempt to do it.

4. The only constructor for a user-defined class is the default (parameterless) constructor. The built-in classes do not have constructors (e.g., the syntax `Int()` is illegal). This is enforced in the front end.

5. Variables in Pyth are all initialized (in Python, they have an "uninitialized" state). Variables whose static type is Int are initialized to 0, and all others are initialized to None.

6. Methods and functions defined by **def** are constants, and may not be modified.

7. The only attributes in a class (or built-in type) are those defined in its definition. Unlike Python, Pyth does not permit defining attributes dynamically by assignment.

8. Pyth allows only "downward funargs" for function values that are not nested immediately inside the global environment. The indicated statements below, legal in Python, are illegal in Pyth:

   ```
   a = ...
   def q (x):
       global a
       def r (y): return x + y
       a = r     # ILLEGAL
       b = r     # OK
       b = [r]   # ILLEGAL (enforced by run-time code)
   ```

```
b = (r, ) # ILLEGAL (enforced by run-time code)
b = { 1:r } # ILLEGAL (enforced by run-time code)
b = { r:1 } # ILLEGAL (enforced by run-time code)
h (r)     # OK (h is some function)
a = q     # OK
b = [q]; b = (q, ); b = {1:q}; b = {q:1}  # OK
return r  # ILLEGAL
return q  # OK
```

The indicated statements are illegal because they attempt (or may attempt) to store or return a function value that contains a non-null environment pointer in a place that outlives that environment. The test is unduly conservative, but easy to implement. The lines that say "enforced by run-time code" above are checks that we make for you. For the others, when assigning to a global variable, or to a variable attribute, your generated code must check that the assigned value is not a function with a non-null environment pointer (that is, the tag field of the assigned value in this case must be ≤ 3).

The reason for this restriction is that (as we covered in lecture some time ago) the need to provide full function closures requires allocating (at least some) local variables on the heap: both a considerable complication in code generation and a considerable expense at execution. It's true that additional static analysis can determine when the expense is unnecessary, but we figure you already have enough to do.

9. Pyth allows *bound methods* only in calls. A bound method is essentially a pair consisting of an object and a method. The following program is legal in Python, but not Pyth:

```
class A:
   y = 3
   def f (self, x): return x + self.y
   g = lambda self, x: x + self.y  # Same thing, but g is variable

h1 = A.f  # OK
h2 = A.g  # OK
r = A()
print r.f (2), r.g (2), h1 (r, 2), h2 (r, 2)  # All OK
h3 = r.f  # ERROR in Pyth
h4 = r.g  # ERROR in Pyth
```

The front end will prevent the abuses in the assignment to `h3`, but not `h4` (since it does not know that `r.g` is a function). Your code must check attribute references that don't immediately result in calls.

10. In Pyth, it is (now) illegal to define a variable attribute in a subclass if there is any attribute with the same name in a superclass. It is illegal to define an attribute in a subclass via **def** if there is a variable attribute (i.e., defined by assignment) of that name in a superclass. It is illegal to override a method (defined via **def**) with a method having a different number of arguments, a different return type, or different parameters types, aside from the type of the first parameter. For example, the indicated lines below are illegal:

```
class A:
   def h (self, x): ...
   h: (A, Int) -> Int
   x = 3
class B(A):
   x = 3   # ILLEGAL
   h = 2   # ILLEGAL
class C(A):
   def x (self): ...   # ILLEGAL
   def h (a, b): ...   # OK
   h: (C, Int) -> Int
class D(A)
   def h (self): ...   # ILLEGAL
class E(A)
   def h (self, x): ...   # ILLEGAL without type declaration
```

These checks are all performed in the front end; you may assume that the program your code generator receives is legal.

11. Pyth Int values are 32-bit integers. Operations on Ints have the same semantics the corresponding operations in Java (in particular, arithmetic is modulo 2^32).

12. Pyth methods always return a value. The front end will insert an explicit return to insure that this happens. If the static type returned by the method is Int, the default return value is 0; otherwise it is `None`.

2 Values

Values (and thus variables) in Pyth are 8 bytes long and consist of a tag and data[1]:

```
typedef struct PythValue PythValue;
typedef PythValue (*PythFunctionPtr)(void* staticLink, ...);

#define INT_TAG    1
#define OBJECT_TAG 3
#define GLOBAL_FUNCTION_TAG 0

struct PythValue {
    union {
        int intValue;
        PythObject* objectPntr;
        PythFunctionPtr funcPntr;
    } data;
    int tag;
};
```

[1] If you don't know C, now would be an excellent time to learn!

The `tag` field indicates what general class of value a `PythValue` contains, and thus which member of the union is valid:

INT_TAG indicates that the value is an integer, and the `data.intValue` field is valid.

OBJECT_TAG indicates that the value is some kind of object pointer either to an object having a built-in type or a user-defined class, and the `data.objectPntr` field is valid.

GLOBAL_FUNCTION_TAG indicates that the value is a pointer to a global function, and that the `data.funcPntr` field is valid.

Any other value of the `tag` field indicates an environment pointer (thus, to test that something is a non-global function, test to see if its `tag` is greater than 3). Environment pointers are always evenly divisible by 4. As described in Lecture #25, each non-zero environment pointer is a static link—the frame pointer for an instantiation of the lexically enclosing function.

Added 1 May 2005

Finally, I should mention one annoying technical point. In the header file `pyth.h`, you will see that functions that, in this document, return a `PythValue`, in actuality return something called a `NativePythValue`. This is just a kludge to get the C compiler to return these 8-byte values in registers rather than using the usual method for returning something described as a C **struct**. Notionally, `NativePythValue` is just another term for `PythValue`.

Added 4/28/2005

3 Objects, Classes, and Type Descriptors

All objects with user-defined types consist of a *type tag*—which is a pointer to a *type descriptor*—and zero or more `PythValues` containing the variable attributes of the object[2]:

```
typedef struct PythObject PythObject;

struct PythObject {
   PythDescriptor* typeTag;
   PythValue attrs[0];
};
```

Constant attributes (defined by **def**) are stored in the type descriptor). The attributes start with all attributes inherited from the parent, in the same order.

A type descriptor contains various information about a type that is used for a variety of purposes:

- To implement method calls;
- To give the garbage collector necessary information; and
- To allow fetching of attributes in objects whose static type is `Any`.

You must create type descriptors for all types, including the pre-defined ones (just follow the native class descriptors).

[2]The definition of `attrs` as a 0-length array is a GCC extension to C, used to indicate arrays whose length is a run-time value, which officially cannot be represented in C. The attributes of the object pointed to by `x` are `x->attrs[0]`, `x->attrs[1]`, etc., as you'd expect. The length of the array of attributes is stored in the type descriptor.

```
typedef struct PythDescriptor PythDescriptor;
typedef struct PythAttributeDescriptor PythAttributeDescriptor;

struct PythDescriptor {
   /** The name of this class (pointer to a null-terminated string) */
   char* className;
   /** The descriptor of this class's parent's descriptor, or 0 if
    *  there is no parent class. */
   PythDescriptor* parent;
   /** The length of an object of this type, in bytes. */
   int objSize;
   /** The total number of variable attributes of this type, including
    *  those inherited from the parent. */
   int numVarAttrs;
   /** The total number of constant method attributes (defined by def),
    *  including those inherited from the parent. */
   int numMethods;
   /** The table of all attribute names and locations, for use by the
    *  run-time library in fetching attributes from objects of
    *  static type Any (a pointer to an array of descriptors). */
   PythAttributeDescriptor (*attrs)[0];
   /** The method table, used for calling known methods. */
   PythFunctionPtr methods[0];
};
```

The `methods` array starts with the method pointers for all inherited methods, in the same order they appear in the parent class's table. Likewise, the array pointed to by the `attrs` field should contain entries for the attributes inherited from the parent as well as those for the class, although order is irrelevant.

A `PythAttributeDescriptor` is an example of *reflection* in the Pyth language—a data structure that provides information about the original source program.

Added 4/25/2005

```
typedef union PythType PythType;

/** A pointer to a type descriptor.  The first word of type descriptor
 *  is a null char* pointer if funcDesc is valid. */
union PythType {
   /** A pointer a class descriptor for the type. */
   PythDescriptor* objDesc;
   /** Pointer to a function-type descriptor for the type. */
   PythFuncDescriptor* funcDesc;
};

enum AttributeKind {
  /** A variable attribute, stored in the object, whose static type
   *  is not a function type. */
  VAR_ATTRIBUTE = 0,
```

```
  /** A variable attribute, stored in the object, whose static type is
   *  a function type. */
  VAR_FUNC_ATTRIBUTE = 1,
  /** A method attribute, stored in the PythDescriptor */
  METHOD_ATTRIBUTE = 2
};

struct PythAttributeDescriptor {
    /** Name of the attribute, a pointer to a null-terminated string. */
    char* name;
    /** Where attribute is stored. */
    enum AttributeKind attrKind;
    /** Byte offset of attribute within the object or .methods array. */
    int offset;
    /** Static type information about the attribute. */
    PythType type;
};
```

For a built-in type named C, generate an assembler label `__typ_`C for its descriptor, and give it external linkage:

```
class Int:                  .globl
   ...       produces    __typ_Int:
                             Contents of Int's type descriptor
```

You can use any naming convention you want for the descriptors of user-defined types (which do not need external (`.globl`) linkage). We'll just use the same one in our example for clarity.

For each user-defined class, C, there is one distinguished object of that class, called the *class exemplar*. Your code must allocate it in initialized static storage, and define it to contain the `typeTag` value, plus default values for all attributes. In this document, we'll give it the label `__obj_`C, but the label you choose is arbitrary. It is this object that is referenced when creating new objects and by references to class methods and variables, as in

```
class A:
   x = 3    # Assigns to the x defined in the class exemplar
A.x = 17    # Same here.
```

4 Built-in Types

Again, you create the type descriptors for built-in classes just as for ordinary classes, but give them external linkage and do not create exemplars for them. They have no variable attributes.

There is only one value of type `Unit`. It has a tag field of 3 (OBJECT_TAG) and an `data.objectPntr` of 0. There is no associated object allocated in memory that corresponds to it.

Objects of the other built-in types start with an ordinary type tag, but the rest of their contents is implementation-defined. They have no program-accessible variable attributes. They may be created and manipulated entirely by their methods, so you don't really need to know the contents, except for Strings, whose layout you need to translate into string literals. This is easily shown by example. The string literal `"Hello, world"` may be translated

```
        .text
        .align 4
    some local label:
        .long  __typ_String
        .long  12  # Length
        .ascii "Hello, world"
```

5 Functions, Methods, and Function Descriptors

Besides their declared parameters, all non-native Pyth functions and methods take one extra parameter, the static link (see the on-line notes for Lecture #25). It is always the first parameter (topmost on the stack). Your code must pass this in every call.

All Pyth functions maintain a frame pointer, and must push the previous value of the frame pointer and establish a new one as their first instructions, using the usual sequence:

```
pushl %ebp
movl %esp,%ebp
```

undoing this on return with[3]:

```
leave
ret
```

Local variables, compiler temporaries, and parameters to function calls all go below (at lower addresses than) the saved frame pointer on the stack. We'll call this area (between the saved frame pointer and the top of the stack) *local storage.* You can do what you want with this area (including expanding or contracting it) as long as the following is true just before each function call:

- Local storage consists of an integral number of PythValues, plus (possibly) a static link at the top of the stack.
- Each of these PythValues that has a tag of OBJECT_TAG contains a pointer to a valid PythObject.

Together, these make it possible for the garbage collector to find all roots on the stack.

All ordinary functions and native functions return a PythValue. The semantic phase will insure that functions always exit by returning some value, possibly None; you don't have to worry about it. Return the tag part of the value in the `%edx` register and the data part in the `%eax` register.

The arguments passed to a function must always conform to that function's static type. In cases like these:

```
def f (x, y): return x + y
def g (x, y): return x + y
g: (Int, Int) -> Int
print f(3,4), g(3,4)
```

[3]Where there's a 'leave' there must be an 'enter,' you might think. Quite true, and it does what you'd expect and much more. Turns out the `pushl, movl` sequence is shorter, however—a wonderful example of CISCness.

there is no problem, since the compiler knows that `f` and `g` take two parameters and that the types of the actual parameters (Int) is a subtype of the formals (`Any` and `Int`, respectively). But in this case:

```
r = lambda x, y: x + y
print r(3,4)
```

that static type of `r` is `Any`, and the run-time system must decide whether it is legal to call `r`. It can tell from the tag of `r` that it holds a function, but it also needs to know the number and types of the parameters.

To provide this information, we place a pointer to a `function descriptor` just before the first instruction of each Pyth function and method. This is the same data structure pointed to in a `PythAttributeDescriptor` to give the static type of a method:

```
typedef struct PythFuncDescriptor PythFuncDescriptor;

struct PythFuncDescriptor {
    /** Always 0, tagging this as a function descriptor. */
    int magic;
    /** Name of this function (points to a null-terminated string).
     *  Used only for tracebacks.  May be null. */
    char* name;
    /** Number of parameters */
    int numParams;
    /** Pointer to type descriptor for the return type. */
    PythType returnType;
    /** Pointers to type descriptors for the formal parameter
     *  types, from first to last in order. */
    PythType formals[0];
};
```

Added 4/25/2005

Again, the size of the `formals` array depends on `numParams`, and so we can't really write it in C, and have used the 0-length array extension of GCC.

6 Native Methods

Native methods are pure C functions. The `.funcName()` child contains the name of the function to call. They obey the standard calling conventions, but differ from Pyth functions in that you do *not* push a static link on the stack for them.

7 The Main Program

The Pyth main program is an ordinary function. Since it will be called from the run-time system, its label, `_pyth_main`, must be declared to have external linkage (with `.globl`. It does not need a pointer to a function descriptor before it, since it is not available to a Pyth program as a function value.

The main program contains all code that is not contained in a lambda or **def**, both inside and outside class definitions. Any variables declared at the outer level are allocated in static storage (the `.data` section), so that the main function has no local variables in the usual sense.

The first duty of the main program is to inform the garbage collector of the addresses of all statically allocated variables, including all exemplar objects (see §3) using two run-time functions provided for that purpose (see §9). Call `__registerVar` with the address (not the contents!) of each global variable that might contain an object pointer (you may, but need not, skip those whose static type is Int, Unit, or a function type). Call `__registerObj` with the address of the exemplar for each user-defined class. Failure to do this will cause bizarre errors when the garbage collector frees storage that is still active.

8 Front-end Modifications to the AST

The front end (lexer, parser, static semantic analyzer) that you'll be using make several modifications to the tree. The idea behind them is to simplify the code-generator's task.

Casting. The front end will wrap expressions whose value must be downcast (i.e., cast to a subtype) in one of two new kinds of node, `CastNode` or `FuncCastNode`, the latter being for casting to function types and the former for casting to all other types. These correspond directly to two run-time operations described in §9. You will never need special cases for assignments, function arguments, returns, or other contexts that require a specific type of value that is not guaranteed by the known (static) type.

Method Calls. A method call, $E.F$`(...)` is converted to a MethodCallNode (new since Project #2), if E represents a value as opposed to a type. If E represents a type, the call is represented by a plain CallNode. This node gives immediate access to E, F, and the argument list. The classes BinopNode and UnopArithNode now extend MethodCallNode rather than CallNodes. Added 4/18/2005

Unknown Attribute References. Attribute references, `x.y`, that are not used as the function in a call and in which `x` is of static type `Any`, require a run-time look-up of the attribute name. All such references will be replaced by native function calls to a function named `__getattr__`, whose static return type is `Any`. Therefore, you won't need to do anything special for these cases, and can assume that the attribute selections left in the tree are all to statically known attributes.

Unknown Function and Method Calls. A call `r(3,4)`, above, where `r`'s static type is `Any`, is converted to a call on the native method `__callfunc__`, producing code equivalent to Moved 4/29/2005

```
native "__callfunc__" (r, 2, 3, 4)  # 2 is the number of parameters.
```

Likewise, $X.Y(E_1, \ldots, E_n)$, where X has static type `Any` (and the declaration attached to Y is unknown), use the native method `__callattr__`, producing the equivalent of

```
native "__callattr__" ("Y", n+1, X, E1,...)
```

Returns. All methods and functions are guaranteed to terminate by returning a value (which is `None` for a plain **return** statement or for a function that simply "falls off the end."). The tree is modified as needed to make this guarantee.

Assignments to Multiple Targets. To make life easier for you, the front end converts: Added 4/21/2005

```
x, y, z = S
```

into something like this:

```
TEMP = S
TEMP: Sequenceable
x = S.__get__ (0)
y = S.__get__ (1)
z = S.__get__ (2)
```

where `TEMP` is a new local variable introduced by the compiler. Here, `Sequenceable` is a new abstract native superclass of `Dict`, `String`, `Tuple`, and `List` that defines the `__get__` and `__len__` operations. As a result, you will not have to handle multi-target assignments.

Assignments to Unknown Attributes, Selections, and Slices. The assignments Added 4/21/2005

```
x[i] = 3
x[i:j] = [1,7]
```

introduce a slight problem: the left-hand sides are represented as method calls or (when `x`'s type is unknown) as native function calls, both of which are normally unassignable. Similarly, in

```
x.y = 3
```

where `x` has type `Any`, the left-hand side would normally be converted to a native method call (see Unknown Attribute References above). To deal with this, we introduce a small kludge. These three cases are rewritten as if they had been

```
x.__setitem__ (i, 3)
x.__setslice__ (i, j, [1,7])
native "__setattr__" ("y", x, 3)
```

(and the first two might then turn into native method calls if the type of `x` is unknown). As a result, you need not worry about these cases, since they'll be converted into other kinds of statements.

9 Run-Time Support Functions

The Pyth run-time system provides a number of functions that your code can call to perform necessary functions. They use the normal calling conventions for C functions. You do not push a static link for them. As indicated in the comments, below, some also vary from the rules given for what the stack must look like before a call (for example, the argument to `__createObject` is a single 4-byte pointer, not an 8-byte PythValue.

```
/** Create a new object consisting of a copy of the object pointed
 *  to by EXEMPLAR (one of the __obj_... objects created for each
 *  class definition). */
PythObject* __createObject (PythObject* exemplar);



/** Checks that VAL's dynamic type is a subtype of TYPE, and
 *  cause an error if not.  The contents of VAL on the stack are
 *  guaranteed not to be disturbed, so that if the value you wish to
 *  check in on top of the stack, you can simply push the descriptor,
 *  call __cast, and pop the descriptor.  Use this function to
 *  implement the CastNode semantics that the front end inserts. */
void __cast (PythType type, PythValue val);

/** Cause an error, terminating the program with the given MSG (a
 *  null-terminated string).  */
void __pythError (char* msg);

/** Print the N trailing values (all PythValues) on the standard
 *  output. */
void __print (int n, ...);

/** Print the N trailing values (all PythValues) on the standard
 *  output, followed by a newline. */
void __println (int n, ...);

/** Create a list of the N items (all PythValues) contained in the
 *  trailing arguments. */
PythValue __consList (int n, ...);

/** Create a tuple of the N items (all PythValues) contained in the
 *  trailing arguments. */
PythValue __consTuple (int n, ...);

/** Create a Dictionary initialized with the N pairs of PythValues in
 *  the trailing parameters.  For example, if X is the PythValue
 *  corresponding to the String "Hello", and Y is the PythValue
 *  corresponding to the Int 42, then __createDict (1, X, Y)
 *  returns { "Hello" : 42 }. */
PythValue __consDict (int n, ...);

/* Note: Strings do not need a __cons... method, since they may be
 * constructed directly in static memory. */

/** Returns the constant 1 if X is a "true" value, and 0
```

Changed 4/25/2005

Added 4/29/2005

```
otherwise.
 *  false values are all those for which either
 *<pre>
 *     + X.data contain 0 (the integer 0, the value None, and null
 *       functions), or
 *     + X denotes a 0-size string, list, tuple, or dictionary.
 *</pre>
 */
int __isTrue (PythValue x);

/* The two __registerXXX functions below have no visible effects, but
 * must be called to inform the garbage collector of roots that exist
 * in statically allocated storage. */

/** Inform the runtime system that space pointed to by VALP is
 *  a variable in static storage.  (This has no visible effect,
 *  but is needed to allow the garbage collector to find all roots).  */
void __registerVar (PythValue* valp);

/** Inform the runtime system that OBJP is a pointer to a
 *  statically allocated exemplar object. */
void __registerObj (PythObject* objp);
```

10 Example

Let's translate the following Pyth code:

```
class A:
   x = 1
   step = 1
   x: Int
   step: Int
   def incr (self):
      self.x += self.step
   def inc (self, y):
      self.x += y;
   inc: (A, Int) -> Unit
class B(A):
   def inc (self, y):
      A.inc (self, y)
      self.step = y
   inc: (B, Int) -> Unit
   z = 1
q = B()
q: A
p = lambda x: x
p: Int -> Any
```

The following is one of many possible translations. The order in which output is produced is largely irrelevant; here we've simply grouped the various kinds of things together, but the assembler won't get confused if you mix it up a little (but do keep constant things in the `.text` segment and variable things in the `.data` segment or there will be trouble. The local names we've chosen (those starting with '.L') are *entirely arbitrary.* In several places, we have used the same string constant multiple times, but there is nothing wrong with duplicating strings, and having only one pointer to each.

```
    .text           # Constant storage
    .globl __typ_Any
__typ_Any:
```
Type descriptor for Any

```
    .globl __typ_Unit
__typ_Unit:
```
Type descriptor for Unit

Type descriptors and definitions for the rest of the standard prelude...

```
    .align 4        # Just a good idea
__typ_A:
    .long .LS1      # className
    .long 0         # parent
    .long 20        # objSize
           # (type tag, x, and step )
    .long 2         # numAttrs
    .long 2         # numMethods
    .long .LA1      # attrs
# methods array:
    .long .LC1      # A.incr's code
    .long .LC2      # A.inc's code
# End of A's descriptor

.LA1:               # attrs table for class A
    .long .LS2      # name (x)
    .long 0         # attrKind (VAR_ATTRIBUTE)
    .long 4         # offset
    .long __typ_Int # objDesc
    .long .LS3      # name (step)
    .long 0
    .long 12
    .long __typ_Int
    .long .LS4      # name (incr)
    .long 2         # attrKind (METHOD_ATTRIBUTE)
    .long 0
    .long .LF1      # Function descriptor for A.incr
    .long .LS5      # name (inc)
    .long 2         # attrKind (METHOD_ATTRIBUTE)
    .long 4
    .long .LF2      # Function descriptor for A.inc
# End of A's attribute table

__typ_B:
    .long .LS6    # "B"
    .long __typ_A   # parent
    .long 28        # objSize
              # (type tag, x, step, and z)
    .long 3         # numAttrs
    .long 2         # numMethods
    .long .LA2      # attrs
# methods array
    .long .LC1      # A.incr's code
    .long .LC3      # B.inc's code
# End of B's descriptor

.LA2:
```
copy of A's attr table, up to but not including inc, followed by...

```
    .long .LS5      # name (inc)
    .long 2         # attrKind (METHOD_ATTRIBUTE)
    .long 4
    .long .LF3      # Function descriptor for B.inc
    .long .LS7      # name (z)
    .long 0         # attrKind (VAR_ATTRIBUTE)
    .long 20        # offset
    .long __typ_Any # objDesc
# End of B's attribute table

.LF1:     # Function descriptor for A.incr
    .long 0         # flag this as a function type.
    .long .LS4      # name (incr)
    .long 1         # numParams
    .long __typ_Any # returnType
    .long __typ_A   # type of parameter

.LF2:     # Function descriptor for A.inc
    .long 0         # flag this as a function type.
    .long .LS5      # name (inc)
    .long 2
    .long __typ_Unit # returnType
    .long __typ_A
    .long __typ_Int

.LF3:     # Function descriptor for B.inc
    .long 0         # flag this as a function type.
    .long .LS5      # name (inc)
    .long 2
    .long __typ_Unit # returnType
    .long __typ_B
    .long __typ_Int

.LF4:     # Function descriptor for p's type.
    .long 0         # flag this as a function type.
    .long 0          # no name
    .long 1
    .long __typ_Any
    .long __typ_Int

.LF5:     # Function descriptor for lambda.
    .long 0         # flag this as a function type.
    .long 0          # no name
    .long 1
    .long __typ_Any
    .long __typ_Any


.LS1:
    .string "A"
.LS2:
    .string "x"
.LS3:
    .string "step"
.LS4:
    .string "incr"
.LS5:
    .string "inc"
.LS6:
    .string "B"
.LS7:
    .string "z"

    .align 4
    .long .LF1
.LC1:               # A.incr
    code for A.incr

    .align 4
    .long .LF2
.LC2:               # A.inc
    code for A.inc

    .align 4
    .long .LF3
.LC3:               # B.inc
    code for B.inc
```

Added 4/25/2005

Added 4/25/2005

Added 4/25/2005

Added 4/25/2005

```
     .align 4
     .long .LF5
.LC4:                # Anonymous lambda
     code for lambda x: x

     .align
     .globl __pyth_main
__pyth_main:
     code to initialize A.x, A.step,
     B.z, q, and p

# Initialized static storage
     .data
     .align 4
__obj_A:   # Exemplar for A
     .long __typ_A
     .long 0        # A.x init value
     .long 1        # A.x is type Int
     .long 0
     .long 1        # A.step is Int

__obj_B:   # Exemplar for B
     .long __typ_B
     .long 0        # B.x
     .long 1        # B.x tag
     .long 0        # B.step
     .long 1        # B.step tag
     .long 0        # None
     .long 3        # B.z tag

.LV_q:               # global q
     .long 0        # None
     .long 3
.LV_p:               # global p
     .long 0        # None
     .long 3        #
```