The VSOP Manual

Cyril SOLDANI, Pr. Pierre GEURTS

March 23, 2016

Contents

1 Introduction

The *Very Simple Object-oriented Programming* language (or VSOP) is, as its name implies, a simple object-oriented language. It is simple enough that a compiler for it can be developed in a short amount of time, but still retains some features of modern programming languages. VSOP is inspired from COOL, the *Classroom Object-Oriented Language* by Alex AIKEN *et al.*

Here are the more salient features of VSOP. Some of them will be explained in more details later in this document, some during the lectures.

- VSOP is meant to be a *general-purpose* programming language. *I.e.* it is not specialized to a specific domain like TEX or Prolog, but should allow to write various kinds of programs easily.
- With features such as automatic memory management and class-based dispatch, VSOP is a somewhat *high-level* language.
- VSOP is an *object-oriented* language. Programs are structured as a set of user-defined *classes*. Each class specifies both a data structure by the way of member *fields*, and operations on that data structure, by the way of member *methods*. This provides information hiding and encapsulation.

  VSOP also supports single-parent *inheritance*.
- VSOP is an *expression-based* language. Nearly all constructs in VSOP are *expressions*, with a type and a value. *E.g.* the expression

  ```
  if cond then e1 else e2
  ```

  has a type and a value, which will be equal to the value of `e1` or the value of `e2`, depending on the runtime value of `cond`. You could then write, *e.g.*

  ```
  someFunction(if cond then e1 else e2)
  ```

  It is thus more similar to C's ternary operator `cond ? e1 : e2` than to C's *if-then-else* construct.

  Most other VSOP constructions are also expression.
- VSOP is *statically typed*, *i.e.* the compiler checks the types of expressions at *compile-time*. This ensures that there will be no type errors at runtime, and allows efficient code generation. It is also somewhat *strongly typed*, by providing no mechanisms to work around the type system, like unsafe casts.
- VSOP is *explicitly typed*, *i.e.* the developer must annotate its code with typing information. While it is certainly a burden, it can help to make programs more legible, and allows the compiler to do a simple *type checking* instead of a much more involved *type inference*.
- VSOP is a *sequential* language. It has no provision for *concurrent* programming, although multi-threaded programs are still possible through library extensions.
- VSOP supports automatic memory management through *garbage collection*.
- VSOP has no support for error handling, like for example an *exception* mechanism.
- VSOP is a *strict*, or *eager* language, *i.e.* function arguments are evaluated before calling a function, and expressions are computed before being stored in a variable or a class field. This is by opposition to a *lazy* language such as Haskell where expressions will generally only be computed if their value is actually needed.
- Finally, VSOP has no support for *polymorphism* (like C++ templates or Java generics), other than *subtyping* (*i.e.* allowing to use a child class where one of its ancestor is expected).

The remainder of the document is organized as follows:

- Section 2 describes the lexical structure of VSOP.
- Section 3 describes the language syntax.
- Section 4 describes the semantics associated to the language, how it is typed and evaluated.
- Section 5 discusses a set of suggested extensions of the basic VSOP language.

2 Lexical Structure

This section describes how the source code character stream should be transformed into VSOP tokens. All token definitions are given in EBNF notation, assuming the following definitions:

```
lowercase-letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
                 | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t"
                 | "u" | "v" | "w" | "x" | "y" | "z";
uppercase-letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
                 | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
                 | "U" | "V" | "W" | "X" | "Y" | "Z";
letter = lowercase-letter | uppercase-letter;
bin-digit = "0" | "1";
digit = bin-digit | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
hex-digit = digit | "a" | "b" | "c" | "d" | "e" | "f" | "A" | "B" | "C" | "D"
          | "E" | "F";
```

Lexical ambiguities are resolved through the longest prefix match rule. *E.g.* the input string `"<="` is to be interpreted as token `lower-equal` rather than token `lower` followed by token `equal`, while `if42` if a valid object identifier notwithstanding the fact that it begins with reserved keyword `if`.

2.1 Whitespace

The following characters are considered whitespace: space, horizontal tabulation, line feed, carriage return and form feed (character to hint a page break).

```
whitespace = { " " | tab | lf | ff | cr };
tab = ? ASCII character 9 ?;
lf = ? ASCII character 10 ?;
ff = ? ASCII character 12 ?;
cr = ? ASCII character 13 ?;
```

Whitespace characters are generally ignored, but do separate adjacent tokens that would otherwise make a single token. *E.g.* `someFun42` is a single identifier while `someFun 42` gives two tokens, the identifier `someFun` followed by the literal `42`.

While line feeds are ignored (*i.e.* they are not returned as part of a token), they still play a role in single-line comments and strings, as explained in sections 2.2 and 2.7.

2.2 Comments

Single-line comments are introduced by `//` and continue up to the next line feed.

Multiple-line comments are introduced by `(*` and continue up to the corresponding `*)`. Contrarily to most C-derived languages, where comments do not nest, these comments may be *nested* arbitrarily. The following code is valid, and contains only a comment:

```
(* Here is a comment. (* Valid nested comment. *) Still commented. *)
```

A multi-line comments should be explicitly terminated, so that reaching end-of-file while inside such a comment is an error in VSOP. However, one can end a single-line comment with end-of-file instead of a line feed.

2.3 Integer Literals

An integer literal is a sequence of one or more digits, optionally prefixed by `0x` for hexadecimal numbers, or `0b` for binary numbers. The default radix when no prefix is used is 10.

```
integer-literal = digit { digit }                 (* base-10 number *)
                | "0x" hex-digit { hex-digit }  (* base-16 number *)
                | "0b" bin-digit { bin-digit }; (* base-2  number *)
```

Rather than interpret a number such as `0xabcdefgh` as integer literal `0xabcdef` followed by object identifier `gh`, it is an error in VSOP. An integer literal followed by an identifier should use some whitespace to separate the two tokens. The same holds for binary numbers, `0b01110047` is an error, not `0b011100` followed by 47. More precisely, a number starting with `0b` or `0x` should continue as long as it is followed by alphanumeric characters (*i.e.* letters or digits). For decimal numbers, such errors are less likely and input such as `42x` should be interpreted as 42 followed by identifier `x`.

2.4 Keywords

The list of keywords in the VSOP language is given in listing 1. Keywords are case-sensitive.

```
and       extends    isnull    then
bool      false      let       true
class     if         new       unit
do        in         not       while
else      int32      string
```

Listing 1: VSOP reserved keywords.

2.5 Type Identifiers

In VSOP, user-defined types always begin with an uppercase letter, and can contain letters, digits and underscores.

```
type-identifier = uppercase-letter { letter | digit | "_" };
```

Primitive built-in types like `bool` begin with a lowercase letter, but are keywords.

2.6 Object Identifiers

In VSOP, all identifiers that are neither keywords, nor type names denotes *objects*. Those identifiers begin with a lowercase letter, and can contain letters, digits and underscores.

```
object-identifier = lowercase-letter { letter | digit | "_" };
```

Note that the identifier `self`, although it has a special meaning (see section 4), is treated like any other object identifier as far as lexical analysis and parsing are concerned.

2.7 String Literals

Strings are sequences of characters enclosed inside a pair of double-quotes ("). A string cannot contain the null character, a line feed, or the end-of-file. All other characters are interpreted literally, with the exception of the backslash (\) which introduce a character escape sequence. All possible character escape sequences are

`\b`	backspace
`\t`	horizontal tabulation
`\n`	line feed
`\r`	carriage return
`\"`	double-quote (**not** ending the string)
`\\`	backslash
`\xhh`	character with byte value `hh` in hexadecimal.

A \ not followed by one of the above escape sequences is an error in VSOP, unless it occurs at the end of a line. A backslash directly followed by a new line and, optionally, a number of spaces and/or tabulations will be ignored, so that

```
"A supposedly very very long str\
     ing."
```

is strictly equivalent to `"A supposedly very very long string."`. This allows to break strings to make them more legible, or enforce some column limit on the source code.

Comments do not occur inside strings, so that the string `"Here comes (* Zorglub *)"` actually contains the word `Zorglub`, as do the string `"Uninterrupted string // Zorglub"`.

All strings should be explicitly terminated, so that reaching end-of-file while inside a string is an error in VSOP.

```
string-literal = '"' { regular-char | escaped-char } '"';
regular-char = ? any valid, non-escaped character as described above ?;
escaped-char = "\" escape-sequence;
escape-sequence = "b" | "t" | "n" | "r" | '"' | "\"
                | "x" hex-digit hex-digit
                | lf { " " | tab }; (* ignored, along with previous "\" *)
```

2.8 Operators

Here is the list of VSOP operators:

```
lbrace = "{";
rbrace = "}";
lpar = "(";
rpar = ")";
colon = ":";
semicolon = ";";
comma = ",";
plus = "+";
minus = "-";
times = "*";
div = "/";
pow = "^";
dot = ".";
equal = "=";
lower = "<";
lower-equal = "<=";
assign = "<-";
```

3 Syntax

The syntax of VSOP is given by the grammar in listing 2. As given, this grammar is ambiguous. It is disambiguated with the precedence and associativity rules in table 1, and the following rules:

- In *if-then-else*, *while*, and *let* expressions, the embedded expressions are taken to be as long as possibly allowed by the grammar. *E.g.*

  ```
  1 + if cond then 0 else 40 + 2
  ```

  is equivalent to

  ```
  1 + (if cond then 0 else (40 + 2))
  ```

  and not

  ```
  1 + (if cond then 0 else 40) + 2
  ```

- An *else* branch should be associated with closest *if-then*. *E.g.*

  ```
  if cond1 then if cond2 then doThis() else doThat()
  ```

  should be interpreted as

  ```
  if cond1 then { if cond2 then doThis() else doThat() }
  ```

  and not as

  ```
  if cond1 then { if cond2 then doThis() }
           else { doThat() }
  ```

Operator	Precedence	Associativity
.	1	left
^	2	right
unary -	3	right
isnull	3	right
*	4	left
/	4	left
+	5	left
-	5	left
<	6	non-associative
<=	6	non-associative
=	6	non-associative
not	7	right
and	8	left
<-	9	right

Table 1: Precedence and associativity rules.

4 Semantics

This section gives the *scoping rules* (*i.e.* describing which entity a name is referring to), the *typing rules* and the *evaluation rules* of VSOP.

VSOP is a *statically typed* language. The compiler typechecks the program at compile time and ensures that no type error can occur at runtime. It does so by assigning a type to every expression, according to the type annotations provided by the developer and the language typing rules.

```
program = class { class };
class = "class" type-identifier [ "extends" type-identifier ] class-body;
class-body = "{" { field | method } "}";
field = object-identifier ":" type [ "<-" expr ] ";";
method = object-identifier "(" formals ")" ":" type block;
type = type-identifier | "int32" | "bool" | "string" | "unit";
formals = [ formal { "," formal } ];
formal = object-identifier ":" type;
block = "{" expr { ";" expr } "}";
expr = "if" expr "then" expr [ "else" expr ]
     | "while" expr "do" expr
     | "let" object-identifier ":" type [ "<-" expr ] "in" expr
     | object-identifier "<-" expr
     | "not" expr
     | expr "and" expr
     | expr ("=" | "<" | "<=") expr
     | expr ("+" | "-") expr
     | expr ("*" | "/") expr
     | expr "^" expr
     | "-" expr
     | "isnull" expr
     | object-identifier "(" args ")"
     | expr "." object-identifier "(" args ")"
     | "new" type-identifier
     | object-identifier
     | literal
     | "(" ")"
     | "(" expr ")"
     | block;
args = [ expr { "," expr } ];
literal = integer-literal | string-literal | boolean-literal;
boolean-literal = "true" | "false";
```

Listing 2: EBNF grammar describing VSOP syntax.

4.1 Classes

All code in VSOP is contained inside classes. All class names also define a type and are globally visible. *I.e.* they can be used anywhere in the input source file, including before the class definition, or inside the class definition itself. Classes cannot be redefined.

A class definition consists of lists of *fields* and *methods*, both of which can be empty. A field of a class `C` specifies a variable that is part of the state of *objects* (also known as *instances*) of class `C`. A method of class `C` is a procedure that manipulates the variables and objects of class `C`.

VSOP provides *information hiding* and *encapsulation* through a simple mechanism. All fields have a scope local to the class, *i.e.* they are *private*. They can only be manipulated through the class' methods. All methods in VSOP have global scope, they are *public*.

Neither a field nor a method can be defined multiple times in the same class. However, a field and a method of a class can have the same name (they resides in different *name spaces*).

As you have seen in section 3, the types of fields, as well as the types of formal arguments and return types of methods must be given explicitly by the developer.

New

An new object of a class is created by using the `new` operator. The expression `new C` creates a fresh object of class `C`. The object can be though of as a record containing a space for each of the fields of class `C`, along with a pointer to a table of its methods. The `new` operator will not only allocate space for the class instance, but will also initialize its fields (and methods table pointer), as explained below.

The type of expression `new C` is `C`.

There is no corresponding `delete` operator in VSOP, which sports *automatic memory management*. Objects which cannot be used anymore will be reclaimed by a *garbage collector*.

Inheritance

If a class `C` *extends* a class `P`, it inherits all the fields and methods of class `P` in addition to its own fields and methods. Class `C` is called a *child* of class `P`. Class `P` is called a *parent* of class `C`.

The inheritance relation is transitive. If a class `C` extends a class `B`, which itself extends a class `A`, then class `C` also extends class `A`.

A child class can be used in any place where one of its parent class can be used. We say that the type of the child class *conforms* to the type of the parent class.

It is illegal to redefine a field of a parent class in a child class. However, a child class can *override* a method of a parent class. The redefined method must have the exact same type (*i.e.* same arguments and return type) than in the parent class, and takes precedence over it (*i.e.* object-oriented dispatch on object of the child class will call the child method).

VSOP only supports *single inheritance*, *i.e.* a class can only extend a single parent class. Moreover, there can be no cycles in the inheritance relation, *i.e.* a class may not extends one of its child class as in the following invalid example:

```
class Bogus extends OtherBogus { (* ... *) }
class OtherBogus extends Bogus { (* ... *) }
```

It is important to distinguish between the *static* type of an expression which is inferred by the compiler at compile time, and the *dynamic* type of that expression during execution. This distinction is necessary because it is not in general possible for the compiler to infer the exact type of values at runtime, due to inheritance (and conformance). The compiler will however ensure that static types are *sound* with respect to all possible dynamic types. See listing 3 for an example illustrating the difference between static an dynamic types.

When not otherwise specified, a *type* will refer to the static type in the remainder of this document.

Dispatch

The dot operator allows to call methods of an object through *object-oriented dispatch*. *E.g.* the code

```
myObject.someMethod(arg1, 42)
```

calls the `someMethod` method of the class of object `myObject` with arguments `arg1` and `42`. There could be several methods `someMethod` in different classes. Which `someMethod` is called depends on the **dynamic** type of object `myObject`.

```
class P { name() : string { "P" } }
class C {
  name() : string { "C" }
  onlyInC() : int32 { (* ... *) }
}
class Other {
  myMethod() : string {
    let p : P <-        // Declared type is P => static type is P.
      if inputInt() = 0 // inputInt() will ask the user for a number.
        then new C      // ‘new C‘ valid here as C conforms to P.
        else new P
    in {
      p.onlyInC(); // Type error. Static type is P, not C. Would be valid
                   // if the user typed 0, but we cannot tell at compile
                   // time.
      p.name() // Dispatch is done using dynamic type.
               // Will return "P" or "C" depending on what the user typed.
    }
  }
}
```

Listing 3: Static types *vs* dynamic types.

Fields

If a field has the optional initializer, it will be executed when a new object is created, and the resulting value assigned to the field. The type of the initializer must conform to the (static) type of the field. The class fields and methods are not yet in scope in the field initializer, as the object is not initialized yet.

If no initializer is present, the field is *default-initialized*. Values of type `int32` will be set to 0, values of type `bool` will be set to `false`, and values of type `string` will be set to `""`. Fields of class types will be set to the special value `null`.

The `null` value is similar to `NULL` in C or `null` in Java. It can be used wherever an object can be used (*e.g.* passed as an argument, stored into a variable, *etc.*), but an attempt to dispatch (*i.e.* call a method) on a `null` value will result in a runtime error. Note that there is no explicit name for `null` values in VSOP. One can only create a `null` value by default-initializing an object variable or field.

One can test whether or not a value is `null` using the `isnull` operator.

Fields are initialized in the order of inheritance, beginning with `Object` down to the object class. Inside a class, fields are initialized in the same order as they appear in the class.

Fields are local to the class they are declared in, and to child classes. They are thus private and can only be manipulated through the class methods, or child class methods.

Fields cannot be named `self`, which has a special meaning (see section 4.2 about identifier).

Methods

A method can have zero or more formal parameters. The identifiers used in the formal parameters list must be distinct.

The result of the method invocation is the result of the evaluation of its body. Within the method body, field names refer to the corresponding fields of the object upon which the method is called. Formal parameter names refer to the corresponding arguments. If a parameter name has the same name than a field, it takes precedence (*i.e.* it *hides* the corresponding field). The special variable `self` refers to the object itself.

The type of the method body must conform to its declared return type.

The Object Class

The pre-defined `Object` class is the default parent class of a class, when its class definition does not explicitly extends a class. As VSOP only support acyclic, single inheritance, it follows that all classes are members of a tree rooted at `Object`, which is the common ancestor of all classes.

The Main Class

A valid VSOP program must provide a `Main` class, with a method `main` with no arguments and returning `int32`. This serves as an entry point to the program.

When the program is run, an object of class `Main` is created, and its method `main` is called. The return value of the method is used as the program exit code[1].

4.2 Expressions

Literals

Literal constants are the simplest expressions. Each literal constant evaluates to its value, and has the following type:

- `true` and `false` are of type `bool`.
- integer literals have type `int32`, and can represent 32-bit signed integers.

[1]As specified by the POSIX standard, the exit code should be 0 if the program executed successfully, and some error code different from 0 if an error occured.

- string literals have type string. A string constant is a contiguous sequence of bytes corresponding to the string characters, followed by a NUL character.
- The only inhabitant of the unit type is () (also called unit). unit is used where void would be used in C, but has one valid value (), which can be passed as argument, stored into a variable, *etc.* The actual representation of unit values does not matter, as the type itself contains all the information. *E.g.* say you expect an argument of type unit. As VSOP is statically typed, you don't need to actually read the argument value at runtime, you already know it can only be (), the only inhabitant of unit.

Identifiers

The names of local variables (introduced by let ... in, see below), formal parameters of methods, self and class fields are all expressions. They evaluate to the current value associated with the local variable, parameter or field, and have the corresponding type.

The *binding* of an identifier references the innermost lexical scope that contains a declaration for that identifier, or to the field of the same name if there is no other declaration. A field can thus be hidden by a formal parameter or local variable, a formal parameter can be hidden by a local variable, and a local variable can be hidden by another local variable declaration with the same name within its scope.

The exception to the previous rule is the identifier self, which is implicitly bound to the current object in every methods, and cannot be hidden (it is an error to declare a field, formal parameter or local variable named self).

Assignments

An assignment of the form <id> <- <expr> first evaluates <expr>, then assign its value to identifier <id>. The type of <expr> must conform to the declared type of <id>. The resulting value of the whole assignment is the value of <expr>.

One cannot assign to self, which always denotes the current object.

Dispatch

Object-oriented dispatch was introduced in section 4.1. More precisely now, consider an expression of the form

```
<expr_0>.<id>(<expr_1>, ..., <expr_n>)
```

To typecheck the dispatch, assuming the **static** type of <expr_0> is P, the compiler will check that class P has a method <id> with <n> formal parameters (<n> can be zero), such that the static type of the i-th actual argument <expr_i> conforms to the type of the i-th formal parameter.

To evaluate the dispatch, <expr_0> is evaluated first. Arguments <expr_1>, ..., <expr_n> are then evaluated from left to right. Finally, assuming <expr_0> has **dynamic** type C, the method <id> of class C is invoked with self bound to the value of <expr_0> and its formal parameters bound to the values of the actual arguments <expr_1>, ..., <expr_n>. The value of the expression is the value returned by the method invocation.

A second form of dispatch

```
<id>(<expr_1>, ..., <expr_n>)
```

is simply a shortcut for a dispatch to self. It is entirely equivalent to

```
self.<id>(<expr_1>, ..., <expr_n>)
```

Conditionals

In a conditional of the form

```
if <cond> then <expr_t> else <expr_e>
```

the condition <cond> must be of bool type.

The types of both branches expr_t and expr_e must agree, which we define as follows:

- If both branches have class type, the types agree and the resulting type of the conditional is the class of the first common ancestor of the two branches.
- If (at least) one branch has type unit, the types agree and the resulting type of the conditional is unit.
- If both branches have another primitive type, the types agree if and only if there are the same. The resulting type of the conditional is the type of both branches.
- Else, the type of both branches don't agree, and it is a typing error.

The evaluation proceeds as follows. The condition <cond> is evaluated first. If its value is <true> the expression <expr_t> is evaluated and <expr_e> is ignored. If the condition value is <false>, expr_e is evaluated and expr_t is ignored. The resulting value of the conditional is () if the conditional has type unit, and the value of the chosen branch otherwise.

A conditional of the form

```
if <cond> then <expr_t>
```

without an else branch is just a shortcut for

```
if <cond> then <expr_t> else ()
```

Loops

In a loop of the form

```
while <cond> do <expr>
```

the condition <cond> must have type bool. The type of <expr> can be any type. The type of the loop is unit[2]

The condition is evaluated before each iteration of the loop. If the predicate is false, the loop terminates and () is returned. If the predicate is true, the body of the loop is evaluated and the process repeats.

Blocks

Expressions in a block are evaluated in the same order as they appear in the block. They can have any type.

The resulting type and value of the whole block is the one from its last expression.

[2] If you wanted to use the type of <expr>, think about the case where the condition is initially false.

Let

A local variable declaration has the form

```
let <id> : <type> [<- <init_expr>] in <body_expr>
```

If an initializer <init_expr> is provided, it is evaluated first and its resulting value is bound to <id> in the body. The type of the initializer must of course conform to the declared type <type>. If no initializer is provided, <id> is bound to the default initializer of <type> (as explained in section 4.1) in the body. The body is then evaluated, and returned as the value of the whole let expression. The type of the let expression is the type of the body.

Note that the scope of the binding is just <body_expr>. After the let expression, <id> takes back its previous binding if it had one (else, it becomes undeclared again).

Finally, note that it is illegal to use self as a bound identifier in a let expression.

Arithmetic, Logic and Comparison Operations

All binary operators first evaluate their left-hand side operand, then their right-hand side operand.

Arithmetic operations, < and <= are only defined on operands of type int32. Arithmetic operations return a value of type int32 (VSOP only supports integer division), according to the usual semantics. < and <= also follow usual semantics, returning a bool.

The equality comparison operator = is special. It can be used on any two values with the same primitive type, returning true if the values are the same and false if they are different. It can also be used between any two class-type objects (not necessarily with the same class). It returns true if both objects have the same memory address, and false otherwise. It is a typing error to try to compare two values with different primitive types, or a value with a primitive type and a value with a class type.

Logical operators act on bool values, and also return a bool, according to the usual semantics.

5 Suggested Extensions

A Examples

These example are here just to give you a feel of the VSOP language. They are not normative, and might change a little by the end of the course.

A.1 Factorial

```
class Main extends IO {
    factorial(n : int32) : int32 {
        if n < 2 then 1
        else n * factorial(n - 1)
    }

    main() : int32 {
        print("Enter an integer greater-than or equal to 0: ");
        let n : int32 <- inputInt() in
        if n < 0 then {
            printError("Error: number must be greater-than or \
                        equal to 0.\n");
            -1
        } else {
            print("The factorial of ").printInt(n).print(" is ");
            printInt(factorial(n)).print("\n");
            0
        }
    }
}
```

A.2 Linked List

```
class List {
    isNil() : bool { true }
    length() : int32 { 0 }
}

(* Nil is nothing more than a glorified alias to List *)
class Nil extends List { }

class Cons extends List {
    head : int32;
    tail : List;

    init(hd : int32, tl : List) : Cons {
        head <- hd;
        tail <- tl;
        self
    }

    head() : int32 { head }

    isNil() : bool { false }
```

```
    length() : int32 { 1 + tail.length() }
}

class Main extends IO {
    main() : int32 {
        let xs : List <- (new Cons).init(0, (new Cons).init(
                                         1, (new Cons).init(
                                         2, new Nil))) in
        print("List has length ").printInt(xs.length()).print("\n");
        0
    }
}
```