University of Arizona, Department of Computer Science

CSc 520 — Assignment 4 — Due noon, Wed April 16 — 10%

Christian Collberg
March 26, 2008

1 Introduction

Your task is to extend LUCA with facilities for object-oriented programming:

1. You should write your interpreter in C or C++ using `gcc`.
2. The interpreter should be named `lucax`. It should read the virtual machine code (in an S-expression format) from standard input.
3. You should test the interpreter on lectura.
4. You should work in a team of two students.
5. If you think it's easier you can remove the garbage collector from assignment 2 and replace `new` with a call to `malloc`.
6. We won't time your code but you should avoid writing your interpreter so that it's obviously inefficient.

The LUCA compiler, code for parsing S-expressions, and some simple test-cases can be downloaded from the class web page: `http://www.cs.arizona.edu/~collberg/Teaching/520/2008/Assignments`.

2 The LUCA Language

Here's is an object-oriented LUCA program `list.luc` that creates a list of integers and reals and prints it out:

```
PROGRAM list;

TYPE ListNode = CLASS
                   [next : ListNode]
                   [METHOD print();
                    BEGIN
                       WRITE 0;
                    END];
TYPE List     = CLASS
                   [first : ListNode]
                   [METHOD add(obj : ListNode);
                    BEGIN
                       obj@next := first@next;
                       first@next := obj;
                    END;
```

```
                    METHOD print();
                    VAR i : ListNode;
                    BEGIN
                       i := first;
                       WHILE i # NIL DO
                          i@print();
                          WRITELN;
                          i := i@next;
                       ENDDO;
                    END
                   ];
TYPE IntNode   = CLASS EXTENDS ListNode
                    [val : INTEGER]
                    [METHOD print();
                     BEGIN
                        WRITE val;
                     END];
TYPE RealNode   = CLASS EXTENDS ListNode
                    [val : REAL]
                    [METHOD print();
                     BEGIN
                        WRITE val;
                     END];
VAR list : List;
VAR in : IntNode;
VAR rn : RealNode;
BEGIN
   list := NEW List;

   in := NEW IntNode;
   in@val := 42;
   list@add(in);

   rn := NEW RealNode;
   rn@val := 42.0;
   list@add(rn);
END.
```

The complete syntax is given in Appendix ??.

3 The Interpreter

These are the object-oriented features of Luca:

- `TYPE T=CLASS [][]` declares a new class `T` with no fields or methods.
- `TYPE U=CLASS EXTENDS T [][]` declares a new class `U` with all of `T`'s (its superclass) fields and methods. `U` may declare additional fields and methods that extend (in the case of fields) or overrides or extends (in the case of methods) the fields and methods of `T`.
- A method `P()` in class `T` will override another method `P` in a superclass of `T`. Both methods must have the same signature.

- `NEW T` allocates a new object of class type `T`.
- `` d`T `` narrows a designator `d` to class type `T` if this is possible (i.e. if `d` is of type `T` or of one of `T`'s subtypes) or aborts with an error-message otherwise.
- `e ISA T` evaluates to `TRUE` if `e` is of type `T` or of one of `T`'s subtypes.
- `x@f` is a designator referencing a field or method `f` in object `x`.
- `SELF` is a reference to the current object. It is only available in methods.
- Assume that `t` is an object of type `T` and `u` is an object of type `U`. The assignment `t := u` is legal if `U` is a subtype of `T`. If that is not the case, a compile-time or run-time error should be generated. Implicit assignments (actual parameters assigned to formal parameters, for example) are also checked in a similar way.
- `NIL` is a new constant which is compatible with all object types.

In addition to what you implemented for assignment 1, your interpreter needs to support

- Non-nested procedures and methods.
- Procedure calls and method invocations.
- The `OBJECT` type.
- Run-time type-checking of casts. Generate the error message

  ```
  ERROR: Can't cast a <type> object to a <type> type.
  ```

  for example

  ```
  ERROR: Can't cast an IntNode object to a RealNode type.
  ```

  You can use either of the two typechecking algorithms described in class.
- Object field references.

4 Virtual Machine Code

From this LUCA program

```
PROGRAM ex;

TYPE C1 = CLASS
            [a:INTEGER]
            [METHOD P(); BEGIN a := 55; END];
TYPE C2 = CLASS EXTENDS C1
            []
            [METHOD P(); BEGIN a := 99; END];

VAR A     : C1;
BEGIN
   A := NEW C2;
```

```
   A@a := 44;
   A@P();
END.
```

the LUCA compiler generates this symbol table:

```
(
   (
      (11 TypeSy OBJECT 0 0 ClassType
         ()
      8 1 1 0)
      (12 ConstSy NIL 0 0 11 1 0)
      (14 ProcedureSy $MAIN 15 0
         ()
         (24)
      1 0)
      (15 TypeSy C1 5 0 ClassType
         (16 17)
      11 1 2 1)
      (16 FieldSy a 4 0 1 1 1 15)
      (17 MethodSy P 5 0 0 18 15)
      (18 ProcedureSy P 5 0
         (19)
         ()
      0 1)
      (19 FormalSy SELF 5 1 15 1 9 0 1 VAL)
      (20 TypeSy C2 8 0 ClassType
         (21)
      15 1 2 1)
      (21 MethodSy P 8 0 0 22 20)
      (22 ProcedureSy P 8 0
         (23)
         ()
      0 1)
      (23 FormalSy SELF 8 1 20 1 9 0 1 VAL)
      (24 VariableSy A 10 0 15 1 0)
   )
```

Notice that each class generates a `ClassType` symbol. For example, `C1` is symbol 15. It has references to the fields and methods (16 and 17) that it contains. In other words, a class type is essentially the same as a record type, except classes can contain methods as well as fields. Notice also that `P` (symbol 18), a method in `C1`, is just a procedure to which the compiler has added an extra parameter, `SELF` (symbol 19). Class `C2` extends `C1` by overriding `P` with its own `P` (symbol 22).

Much of the complexity of supporting object oriented features is handled by the compiler — building a correct symbol table is the main issue!

Here are the generated bytecodes for the two `P` methods:

```
   (
      (info 15 8 6 29 1 14 24)

      (begin 5 18 1 0 15 1 0 P)
```

```
        (apush 5 19 SELF)
        (aload 5 15)
        (ofieldof 5 16 a)
        (ipush 5 55)
        (istore 5)
        (end 5 18 P)

        (begin 8 22 1 0 20 1 0 P)
        (apush 8 23 SELF)
        (aload 8 20)
        (ofieldof 8 16 a)
        (ipush 8 99)
        (istore 8)
        (end 8 22 P)
```

Notice how the compiler turned the statement `a := 55;` into `SELF^.a := 55;`! The `ofieldof` bytecode is similar to the `fieldof`, except there's an extra indirection.

Again, the only difference between a method and a procedure is that the compiler gave the method an extra `SELF` parameter. Also, methods are always invoked through a reference to an object. LUCA's procedures are the same as Java's `static` methods.

```
        (begin 15 14 0 1 9 0 1 $MAIN)
        (apush 12 24 A)
        (tpush 12 20 C2)
        (new 12 20)
        (astore 12 15)
        (apush 13 24 A)
        (aload 13 15)
        (ofieldof 13 16 a)
        (ipush 13 44)
        (istore 13)
        (apush 14 24 A)
        (aload 14 15)
        (icall 14 17 P)
        (end 15 14 $MAIN)
    )
)
```

The semantics of the `new` bytecode has changed since assignment 2. It now takes an argument on the stack, the object template. The new bytecode `tpush` pushes a reference to this template on the stack.

There are two new bytecodes for calling procedures and methods, `call` (for procedures) and `icall` for methods. The only difference is that `icall` takes an additional argument on the stack, `SELF`, that the compiler added. So, `A@P();` turns into

1. push the value of `A` on the stack,
2. look up the right method `P` by looking in the template attached to `A`,
3. do an indirect call to that method.

4.1 Runtime type checking

Here's a program that exercises LUCA's runtime typechecking:

```
PROGRAM ex;

TYPE C1 = CLASS [][];
TYPE C2 = CLASS EXTENDS C1 [][];

VAR A : C1;
VAR B : C2;
VAR x : C1;
BEGIN
   A := B;
   B := A;
   x := B‘C1;
   IF B ISA C2 THEN
      WRITE 99;
   ENDIF;
END.
```

The “back-tick” operator (`variable‘type`) explicitly casts a variable to a particular type. If the cast doesn't work, an error message should be generated and the program terminates. Notice that the compiler will insert implicit typechecks when necessary, for example for the `B := A` statement. Type casts only work for object types. The `ISA` operator checks if an object variable is of a particular type.

The `A:=B; B:=A;` statements in the above example generate this bytecode:

```
   (
      (info 16 8 6 29 3 14 19)
      (begin 16 14 0 3 9 0 3 $MAIN)
      (apush 10 17 A)
      (apush 10 18 B)
      (aload 10 16)
      (astore 10 15)

      (apush 11 18 B)
      (apush 11 17 A)
      (aload 11 15)
      (tpush 11 16 C2)
      (narrow 11)
      (astore 11 16)
```

Notice that `A := B` generates nothing special, but `B := A` inserts a `narrow` operator. This takes an object and a class template as arguments on the stack and does a type check. If the check fails, an error is generated, otherwise `narrow` just returns the object on the stack, and the assignment can proceede.

The assignment `x := B‘C1` generates this convoluted code, mostly because the compiler is retarded:

```
      (apush 12 19 x)
      (apush 12 18 B)
      (adup 12 15)
```

```
        (aload 12 15)
        (tpush 12 15 C1)
        (narrow 12)
        (pop 12)
        (aload 12 15)
        (astore 12 15)
```

You have to add the bytecode `pop` which pops off the top element on the stack, and `adup` which copies the top element on the stack.

Finally, the `isa` instruction works like the other relational operators, it takes an offset to jump to as argument:

```
        (apush 13 18 B)
        (aload 13 16)
        (tpush 13 16 C2)
        (isa 13 2)
        (jmp 13 3)
        (ipush 14 99)
        (iwrite 14)
        (end 16 14 $MAIN)
    )
)
```

The `isa` operator, like `narrow`, takes an object and a class template as arguments. Both, therefore, should call the same internal runtime function to figure out, by walking the class hierarchy tree, if the object is an instance of the particular class. The difference is that `narrow` aborts with an error if the test fails, whereas `isa` just returns false.

4.2 Procedure calls

To implement procedure calls you'll need a stack of activation records, where each record has a region for local variables and one for formal parameters. When the `call` is issued, the actual arguments are (in reverse order) on the stack. Easiest is just pop them off the stack, copy them into a new activation record, and set the PC to the first instruction of the new procedure. You don't need to be able to handle arguments of array or record type, although that's not hard.

Here's an example:

```
PROGRAM P1;
   PROCEDURE Q(X : INTEGER; VAR Y : INTEGER);
   VAR Z : INTEGER;
   BEGIN
      Z := 4;
      X := Z;
      Y := 4;
   END;

VAR A : INTEGER;
VAR B : INTEGER;
BEGIN
   Q(A,B);
END.
```

Note that `B` is passed by reference (like `&B` in C) but `A` is passed by value. This is because of the `VAR` declaration of `Y` in the procedure heading.

There's a `ProcedureSy` symbol for procedures that gives the amount of memory that needs to be reserved in the activation record for arguments and local variables:

```
      (15 ProcedureSy Q 8 0
         (16 17)
         (18)
      1 2)
      (16 FormalSy X 2 1 1 1 9 0 1 VAL)
      (17 FormalSy Y 2 1 1 1 9 1 2 VAR)
      (18 VariableSy Z 3 1 1 1 0)
   )
```

`FormalSy` is like `VariableSy`. Local variables and formal parameters both start with a 0 offset.

Local variables are treated the same as global ones, except they're stored on the stack:

```
      (begin 8 15 2 1 9 2 1 Q)

      (apush 5 18 Z)
      (ipush 5 4)
      (istore 5)

      (apush 6 16 X)
      (apush 6 18 Z)
      (iload 6)
      (istore 6)

      (apush 7 17 Y)
      (iload 7)
      (ipush 7 4)
      (istore 7)

      (end 8 15 Q)
```

Notice how the compiler treats `X` and `Y` differently — `Y` is passed by reference so an extra `iload` is inserted.

The first argument is on top of the stack, i.e. the arguments are pushed in the reverse order that they appear on the argument list:

```
      (begin 15 14 0 2 9 0 2 $MAIN)
      (apush 13 20 B)
      (apush 13 19 A)
      (iload 13)
      (call 13 15 Q)
      (end 15 14 $MAIN)
   )
)
```

Here, `B`'s address is pushed, but `A`'s *value* is pushed. The `call` instruction is responsible for popping the arguments off the stack and passing them to the callee.

Appendix **??** has a complete listing of all the bytecodes.

5 OK, so what do I have to do???

In addition to the instructions you implemented in assignment 1, you should implement the following bytecode instructions:

1. `adup` and `pop` for stack manipulation,
2. `call` for procedure calls,
3. `ofieldof` for dereferencing object fields,
4. `isa` and `narrow` for type checking,
5. `pushnil` which pushes the `NIL` value (0) on the stack,
6. `new` which allocates an object from the heap (just a call to `malloc` followed by an assignment of the template field), and
7. `icall` for method invocations.

You also need to be able to handle the `ClassType`, `MethodSy`, `ProcedureSy`, and `MethodSy` entries in the symbol table.

So, what, do you ask, should I do???? Well, you know by now: 

! Here's one way to go about working this assignment:

1. Implement `pushnil`, `adup`, `pop`. Hey, that was easy!
2. Implement procedure calls, i.e. the `call` bytecode. You need to fix the `apush` instruction to handle pushing local variables and formal parameters on the stack. You have to handle the `ProcedureSy` and `FormalSy` symbol table entries.
3. Fix `new` to call `malloc` and to take the template pointer off the stack.
4. Implement `ofieldof`. It's like `fieldof` except the variable is a pointer, so you have to add an extra level of indirection.
5. Build templates for each class. This involves having to handle the `ClassType` and `MethodSy` symbol table entries.
6. Implement `icall`. This just walks the template pointer from the object that's on the top of the stack and finds the right method to jump to in the template.
7. Build a function `isType(templatePtr,objectPtr)` and implement `narrow` and `isa` to call this function. You can implement `isType` simply by walking up the inheritance tree that the templates form, or you can use the $O(1)$ algorithm we discussed in class.

6 Extension

For an extra 10 points, write a cool object oriented Luca program!

7 Submission and Assessment

The deadline for this assignment is noon, Wed April 16. It is worth 10% of your final grade.

You should submit the assignment electronically using the `Unix` command

```
turnin cs520.4 lucax.c Makefile README
```

.

Your submission *must* contain a file called `README` that states which parts of LUCA your interpreter can handle. Also, list the name of your team, the team members, and how much each team member contributed to the assignment.

Your electronic submission *must* contain a working `Makefile`, and *all* the files necessary to build the interpreter. If your program does not compile "out of the box" you *will* receive *zero* (0) points. The grader will *not* try to debug your program or your makefile for you!

Don't show your code to anyone, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the instructor or the TA.

A The Luca Language

program ::=
  ‘**PROGRAM**’ ident ‘;’ decl_list block ‘.’

decl_list ::=
  { declaration ‘;’ }

declaration ::=
  ‘**VAR**’ ident ‘:’ ident |
  ‘**TYPE**’ ident ‘=’ ‘**ARRAY**’ expression ‘**OF**’ ident |
  ‘**TYPE**’ ident ‘=’ ‘**RECORD**’ ‘[’ { field } ‘]’ |
  ‘**TYPE**’ ident ‘=’ ‘**REF**’ ident |
  ‘**CONST**’ ident ‘:’ ident ‘=’ expression |
  ‘**TYPE**’ ident ‘=’ ‘**CLASS**’ [‘**EXTENDS**’ ident] ‘[’ field_list ‘]’ ‘[’ method_list ‘]’|
  ‘**PROCEDURE**’ ident ‘(’ [formal_list] ‘)’ decl_list block ‘;’

field ::= ident ‘:’ ident ‘;’

method_list ::= method_decl [‘;’ method_list]

method_decl ::= ‘**METHOD**’ ident ‘(’ [formal_list] ‘)’ ‘;’ decl_list block

designator ::= ident { designator’ }

formal_list ::=
  formal_param { ‘;’ formal_param }

formal_param ::=
  [‘**VAR**’] ident ‘:’ ident

actual_list ::=
  expression { ‘,’ expression }

block ::=
  ‘**BEGIN**’ stat_seq ‘**END**’

stat_seq ::=
  { statement ‘;’ }

statement ::=
  designator ‘:=’ expression |
  ‘**WRITE**’ expression |
  ‘**WRITELN**’
  ident ‘(’ [actual_list] ‘)’
  ‘**IF**’ expression ‘**THEN**’ stat_seq ‘**ENDIF**’|
  ‘**IF**’ expression ‘**THEN**’ stat_seq ‘**ELSE**’ stat_seq ‘**ENDIF**’ |
  ‘**WHILE**’ expression ‘**DO**’ stat_seq ‘**ENDDO**’ |
  ‘**REPEAT**’ stat_seq ‘**UNTIL**’ expression |
  ‘**LOOP**’ stat_seq ‘**ENDLOOP**’ |
  ‘**EXIT**’ |
  ‘**GC**’ |
  ‘**READ**’ designator

expression ::=
  expression bin_operator expression |
  ‘(’ expression ‘)’ |
  unary_operator expression |
  real_literal |
  integer_literal |
  char_literal |
  designator | string_literal

designator ::=

ident |
designator '[' expression ']' |
designator '.' ident |
designator '@' ident |
designator '`' ident |
'^' designator'

unary_operator ::=
'−' | '**TRUNC**' | '**FLOAT**'| '**NOT**' | '**NEW**' ident

bin_operator ::=
'+' | '−' | '∗' | '/' | '%' | '**ISA**' | '<' | '<=' | '=' | '#' | '>=' |'>' |'AND' | 'OR'

- The LUCA language is case sensitive.
- Luca has four (incompatible) built-in types: `INTEGER`, `CHAR`, `BOOLEAN` and `REAL`. All basic types are 64 bits wide. These are the type rules for Luca:

Left	Operators	Right		Result
Int	'+', '−', '∗', '/', '%'	Int	⇒	Int
Real	'+', '−', '∗', '/'	Real	⇒	Real
Int	'<','<=', '=', '#', '>=', '>'	Int	⇒	Bool
Real	'<','<=', '=', '#', '>=', '>'	Real	⇒	Bool
Char	'<','<=', '=', '#', '>=', '>'	Char	⇒	Bool
Bool	'**AND**', '**OR**'	Bool	⇒	Bool
	'**NOT**'	Bool	⇒	Bool
	'−'	Int	⇒	Int
	'−'	Real	⇒	Real
	'**TRUNC**'	Real	⇒	Int
	'**FLOAT**'	Int	⇒	Real

- The '#' symbol means "not equal to". `AND` and `OR` have lower precedence than the comparison operators, which in turn have lower precedence than the arithmetic operators.
- Pointers can only be compared with `=` and `#`, not `<`, `<=`, etc.
- LUCA does not allow *mixed arithmetic*, i.e. there is no *implicit conversion* of integers to reals in an expression. For example, if `I` is an integer and `R` is real, then `R:=I+R` is illegal. LUCA instead supports two explicit conversion operators, `TRUNC` and `FLOAT`. `TRUNC R` returns the integer part of `R`, and `FLOAT I` returns a real number representation of `I`. Note also that `%` (remainder) is not defined on real numbers.
- The identifiers `TRUE`, `FALSE`, and `NULL` are predeclared in the language.
- Arrays are indexed from 0; that is, an array declared as `ARRAY 100 OF INTEGER` has the index range `[0..99]`. It is a checked run-time error to go outside these index bounds. The size of the array must be a compile-time constant.
- Assignment is defined for scalars only, not for variables of structured type. In other words, the assignment `A:=B` is illegal if `A` or `B` are records or arrays.
- **READ** and **WRITE** are only defined for scalar values (integers, reals, and characters). String constants can also be written.
- Comments start with a $ and go the end of the line.
- A procedure's formal parameters and local declarations form one scope, which means that it is illegal for a procedure to have a formal parameter and a local variable of the same name.

- Parameters are passed by value unless the formal parameter has been declared **VAR**. Only L-valued expressions (such as '`A`' and '`A[5]`') can be passed to a **VAR** formal.
- Procedures cannot be nested.
- Identifiers have to be declared before they are used.

B LVM – The Luca Virtual Machine

- LVM is a word-addressed machine. Words are 64 bits wide. The size of all basic types (integers, reals, booleans, and chars) is one word.
- Branch offsets for the `ieq, ine, jmp, ...` are also 8 bytes long.
- LVM is a stack machine. Conceptually, there is just one stack and it is used both for parameter passing and for expression evaluation. An implementation may – for efficiency or convenience – use several stacks. For example, in a three stack LVM implementation one stack can be used to store activation records, one can be used for integer arithmetic and one can be used for real arithmetic.
- Execution begins at the (parameterless) procedure named `$MAIN`.
- Large value parameters are passed by reference. It is the responsibility of the called procedure to make a local copy of the parameter. For example, if procedure `P` passes an array `A` by value to procedure `Q`, `P` actually pushes the *address* of `A` on the stack. Before execution continues at the body of `Q`, a local copy of `A` is stored in `Q`'s activation record. The body of `Q` accesses this copy. A special LVM instruction `Copy` is inserted by the front end to deal with this case.
- When a `Call` instruction is encountered the arguments to the procedure are on the stack, with the first argument on the top. In other words, arguments are pushed in the *reverse* order.
- Note that bytecode offsets in the branch instructions `ieq, ine, jmp, ...` are in terms of instruction *numbers*, not byte-offsets in the bytecode. When you read in the instructions and translate them to your own bytecode you must calculate the actual address in the bytecode array.

B.1 The LVM instruction set

Below, t is one of the letters `i,r,c,s,a`, giving the type of the operands of the instruction: `i=integer`, `r=real`, `c=character`, `s=string`, `a=address`.

All instructions have a `Pos` argument which gives the (approximate) line number in the source code from which the instruction was generated. Some instructions also have a `Name` argument which gives the name of the identifier in the source code the instruction operates on. `Name` should not be used by the interpreter — it is there for ease of debugging only. `Pos` can be used to print out informative error messages.

Every instruction shows its effect on the stack. For example, $[L, A] \Rightarrow [L + A]$ means that before the instruction executes there are (at least) two elements on the top of the stack, L and A. L is the top element, A the one below the top. After the instruction has finished executing L and A have been popped off the stack and their sum, $L + A$, pushed.

B.1.1 Procedures

Instruction	Stack
(**info** `Pos Major Minor Instrs Globals Main Symbols)` Information about the LVM file. Always the first instruction. `Major`: The major version number. `Minor`: The minor version number. `Instrs`: The number of instructions in the file. `Globals`: The amount of memory that should be allocated for global variables. `Main`: The symbol number of the `$MAIN` procedure. `Symbols`: The number of declared symbols.	$[\,] \Rightarrow [\,]$
(**begin** `Pos SyNo FormalCount LocalCount Type FormalSize LocalSize Name)` The beginning of a procedure. `SyNo`: The symbol number of the procedure. `FormalCount`: The number of formal parameters. `LocalCount`: The number of local variables. `Type`: For future use. `FormalSize`: The size of formal parameters. `LocalSize`: The size of local variables. `Name`: The name of the procedure.	$[\,] \Rightarrow [\,]$
(**end** `Pos SyNo Name)` The end of the procedure.	$[\,] \Rightarrow [\,]$
(**call** `Pos ProcNo Name)` Call the procedure whose symbol number is `ProcNo`. The arguments to the procedure are on the stack, with the first argument on the top. If an argument is passed by reference it's address is pushed, otherwise its value. Large value parameters are also passed by reference and the called procedure is responsible for making a local copy.	$[\mathrm{arg}_1, \mathrm{arg}_2, \ldots] \Rightarrow [\,]$
(**icall** `Pos ProcNo Name)` The argument on top of the stack is `SELF`. Use it to geto to the object template, and work out which method to call.	$[\mathrm{arg}_1, \mathrm{arg}_2, \ldots] \Rightarrow [\,]$

B.1.2 Arrays, Records, Objects, and Pointers

Instruction	Stack
(indexof `Pos ArrayNo Name`) Compute the address of an array element. The address of the array and the index value are on the top of the stack. The address should be incremented by $I * \texttt{ElmtSz}$, where `ElmtSz` can be found from the array declaration. If I is not within the array's bounds, a fatal error should be generated. `ArrayNo`: The symbol number of the array.	$[A, I] \Rightarrow [A + I * \texttt{ElmtSz}]$
(fieldof `Pos FieldNo Name`) Compute the address of a field of a record. The address of the record is on the top of the stack. The address should be incremented by `FieldOffset`, the offset of the field within the record. `FieldNo`: The symbol number of the field.	$[R] \Rightarrow [R + \texttt{FieldOffset}]$
(refof `Pos RefNo`) A address is on top of the stack. Push the value stored at that address. `RefNo`: The symbol number of the pointer.	$[R] \Rightarrow [R\hat{}\,]$
(ofieldof `Pos FieldNo Name`) Compute the address of a field of a record. The object pointer is on the top of the stack. Follow it to the object and add the field offset. `FieldNo`: The symbol number of the field.	$[R] \Rightarrow [R\hat{}+\texttt{FieldOffset}]$

B.1.3 Loading and Storing

Instruction	Stack
(*t***push** `Pos Val`)	[ ] $\Rightarrow$ [Val]
Push a literal value. **`ipush`** pushes an integer value, **`rpush`** a real, **`cpush`** a character, and **`spush`** a string. Pushing a string means pushing its address. For other types, the value of the constant is pushed.	
(**pushnull** `Pos`)	[ ] $\Rightarrow$ [0]
Push a `NULL`, i.e. 0, on the stack	
(**pushnil** `Pos`)	[ ] $\Rightarrow$ [0]
Push a `NIL`, i.e. 0, on the stack	
(**apush** `Pos SyNo Name`)	[ ] $\Rightarrow$ [`addr(SyNo)`]
Push the address of the local or global variable or formal parameter whose symbol number is `SyNo`.	
(*t***store** `Pos`)	$[L, R] \Rightarrow [\,]$
Store value R at address L. $t \in \{\mathtt{i}, \mathtt{r}, \mathtt{c}\}$.	
(*t***load** `Pos`)	$[L] \Rightarrow [R]$
Push the value R stored at address L onto the stack. $t \in \{\mathtt{i}, \mathtt{r}, \mathtt{c}\}$.	
(**astore** `Pos SyNo`)	$[L, R] \Rightarrow [\,]$
Store value R (a pointer) at address L. `SyNo` is the type of the pointer stored.	
(**copy** `Pos Type Size`)	$[L, R] \Rightarrow [\,]$
Copy `Size` number of words from address L to address R. Currently, the front-end only generates this instruction to make local copies of large value formal parameters.	
(**new** `Pos SyNo`)	$[T] \Rightarrow [P]$
`SyNo` is the symbol number of a pointer type. Allocate a new object of this type on the heap and push the pointer P on the stack. T is the template pointer. Store this in the object header. The memory should be filled with zeroes.	
(**narrow** `Pos`)	$[L, T] \Rightarrow [L]$
L is an object pointer and T a template pointer. Pop T off the stack. If L is not of type T or a subtype of T then generate an error message and exit the program.	
(**adup** `Pos`)	$[L] \Rightarrow [L, L]$
Push a copy of the top element on the stack.	
(**pop** `Pos`)	$[L] \Rightarrow [\,]$
Pop one element off the top of the stack.	
(**gc** `Pos SyNo`)	$[\,] \Rightarrow [P]$
Perform a garbage collection.	

B.1.4 Arithmetic

Instruction	Stack
(*t***add** `Pos`)	$[L,R] \Rightarrow [L+R]$
Pop two elements off the stack and push their sum. $t \in \{\texttt{i},\texttt{r}\}$.	
(*t***sub** `Pos`)	$[L,R] \Rightarrow [L-R]$
Pop two elements off the stack and push their difference. $t \in \{\texttt{i},\texttt{r}\}$.	
(*t***mul** `Pos`)	$[L,R] \Rightarrow [L*R]$
Pop two elements off the stack and push their product. $t \in \{\texttt{i},\texttt{r}\}$.	
(*t***div** `Pos`)	$[L,R] \Rightarrow [L/R]$
Pop two elements off the stack and push their quotient. $t \in \{\texttt{i},\texttt{r}\}$.	
(**imod** `Pos`)	$[L,R] \Rightarrow [L\%R]$
Pop two integers off the stack and push their remainder. I.e. there's no floating point modulus operator.	
(*t***uminus** `Pos`)	$[L] \Rightarrow [-L]$
Pop one element L off the stack and push $-L$. $t \in \{\texttt{i},\texttt{r}\}$.	
(**trunc** `Pos`)	$[L] \Rightarrow [\lfloor L \rfloor]$
Convert L (a real) to an integer by rounding down.	
(**float** `Pos`)	$[L] \Rightarrow [(\texttt{float})L]$
Convert L (a integer) to the clostest corresponding real.	

B.1.5 Input and Output

Instruction	Stack
(*t***write** `Pos`)	$[L] \Rightarrow [\,]$
Write L to standard output. $t \in \{\texttt{i},\texttt{r},\texttt{c},\texttt{s}\}$.	
(**writeln** `Pos`)	$[\,] \Rightarrow [\,]$
Write a newline character.	
(*t***read** `Pos Type`)	$[L] \Rightarrow [\,]$
Read a value from standard input and store at address L. $t \in \{\texttt{i},\texttt{r},\texttt{c}\}$. It's a fatal error if the value read is the wrong type.	

B.1.6 Branching

Instruction	Stack
(*t**op*** `Pos Offset`)	$[L,R] \Rightarrow [\,]$
If L *op* R then goto `PC`+`Offset`, where `PC` is the number of the current instruction. $t \in \{\texttt{i},\texttt{r},\texttt{c}\}$, *op* $\in \{\texttt{eq},\texttt{ne},\texttt{lt},\texttt{gt},\texttt{le},\texttt{ge}\}$, i.e. there are instructions `ieq`, `req`, `ceq`, `ine`, `rne`, etc.	
(**a*op*** `Pos Offset`)	$[L,R] \Rightarrow [\,]$
If L *op* R then goto `PC`+`Offset`, where `PC` is the number of the current instruction. *op* $\in \{\texttt{eq},\texttt{ne}\}$. I.e. the two instruction `aeq` and `ane` compare addresses.	
(**isa** `Pos Offset`)	$[L,T] \Rightarrow [\,]$
L is an object pointer and T a template pointer. If L is of type T or a subtype of T jump to `PC`+`Offset`.	
(**jmp** `Pos Offset`)	$[\,] \Rightarrow [\,]$
Goto `PC`+`Offset`.	

B.1.7 Symbol Tables

The first part of the LVM code is always a symbol table. The format of each symbol is

```
({\em number} {\em kind} {\em name} {\em position} {\em level} \ldots)
```

where ... represents information which is specific to each symbol kind. *number* is a unique number used to identify each symbol. *kind* describes what type of symbol we're dealing with, one of `VariableSy`, `ConstSy`, `EnumSy`, `FormalSy`, `FieldSy`, `ProcedureSy` and `TypeSy`. *name* is the name of the symbol. *level* is 0 for global symbols and 1 for symbols declared within procedures.

The information specific to each symbol is given below. Attributes in *italic font* are standard for all symbols. Attributes in **bold font** are atoms describing the symbol kind. Attributes in `typewriter font` are specific to a particular symbol.

(***number*** **VariableSy** ***name pos level*** `type size offset`)
: This entry represents a declared variable. `type` is the symbol number of the type of the variable. `size` and `offset` are the size (in bytes) and the address of the variable.

(***number*** **ConstSy** ***name pos level*** `type value`)
: This entry represents the value of a constant declaration. For integers, floats, and characters the value is simply the obvious textual representation. For booleans it is the atom `TRUE` or `FALSE`.

(***number*** **EnumSy** ***name pos level*** `type size value`)
: This is only used for BOOLEAN types since this version of LUCA does not allow the declaration of enumeration types.

(***number*** **FormalSy** ***name pos level*** `type size copy offset formalNo mode`)
: This represents a formal parameter of a procedure. `formalNo` is the number of the formal, where the first parameter has the number 1. `mode` is one of `VAL` and `VAR`.

(***number*** **FieldSy** ***name pos level*** `type size offset parent`)
: This represents a field in a record. `type` is the symbol number of the type of the symbol. `parent` is the symbol number of the record type itself.

(***number*** **ProcedureSy** ***name pos level*** `formals locals localSize formalSize`)
: This represents a procedure declaration. `formals` is a list (for example, `"(12 13 14)"`) of the symbol numbers of the formal parameters. `locals` is a list of the symbol numbers of the local variables.

(***number*** **MethodSy** ***name pos level*** `offset proc parent`)
: This represents a method declaration. `offset` is the method's offset in the tempalte, `proc` the actual procedure which should be called, and `parent` the class to which the method belongs.

(***number*** **TypeSy** ***name pos level*** **BasicType** `size`)
: This represents a basic type such as integer or real.

(***number*** **TypeSy** ***name pos level*** **ArrayType** `count type size`)
: This represents an array type. `count` is the number of elements of the array. `type` is the symbol number of the element type.

(***number*** **TypeSy** ***name pos level*** **RecordType** `fields size`)
: This represents a record type. `fields` is the list of symbol numbers of the fields of the record.

(***number*** **TypeSy** ***name pos level*** **RefType** `type size`)
: This represents a reference (pointer) type. `type` is the symbol number of the pointed to type. `size` is always 1 since pointers are one word long.

(*number* **TypeSy** *name pos level* **ClassType** `fields supertype size fieldsize methodsize`)
: This represents a class type. `fields` is the list of symbol numbers of the fields and methods of the class.

(*number* **TypeSy** *name pos level* **EnumType** `size`)
: This represents an enumeration type type. This version of LUCA doesn't have declarations of enumeration types so the only place where this symbol occurs is in the declaration of the standard boolean type.

The following symbols are predeclared by the compiler:

```
(1 TypeSy INTEGER 0 0 BasicType 1)
(2 TypeSy REAL 0 0 BasicType 1)
(3 TypeSy CHAR 0 0 BasicType 1)
(4 TypeSy STRING 0 0 BasicType 0)
(5 TypeSy BOOLEAN 0 0 EnumType
   (6 7)
1)
(6 EnumSy TRUE 0 0 5 0 1)
(7 EnumSy FALSE 0 0 5 0 0)
(8 TypeSy $NOTYPE 0 0 BasicType 0)
(9 TempSy $NOSYMBOL 0 0 8 0 0)
(10 TypeSy $ADDRESS 0 0 BasicType 1)
(11 TypeSy OBJECT 0 0 ClassType
   ()
8 1 1 0)
(12 ConstSy NIL 0 0 11 1 0)
(13 ConstSy NULL 0 0 10 1 0)
(14 ProcedureSy $MAIN 21 0 ...)
```

This has the following consequences:

1. The first symbol declared by the program will get symbol number 15.
2. Integer types are always symbol number 1, reals are symbol number 2, etc.
3. The main program is represented by symbol number 14.

Here is an example of the output from `luca_sem` for a simple program:

```
> cat T.luc
PROGRAM P;
   VAR X : BOOLEAN;
   TYPE A = ARRAY 10 OF CHAR;
   TYPE R = RECORD [x:INTEGER];
   CONST C : INTEGER = 10;
   PROCEDURE P (VAR x : REAL; y: R);
   BEGIN END;
BEGIN
END.
> lc T.luc
(
```

```
(
   (1 TypeSy INTEGER 0 0 BasicType 1)
   (2 TypeSy REAL 0 0 BasicType 1)
   (3 TypeSy CHAR 0 0 BasicType 1)
   (4 TypeSy STRING 0 0 BasicType 0)
   (5 TypeSy BOOLEAN 0 0 EnumType
      (6 7)
   1)
   (6 EnumSy TRUE 0 0 5 0 1)
   (7 EnumSy FALSE 0 0 5 0 0)
   (8 TypeSy $NOTYPE 0 0 BasicType 0)
   (9 TempSy $NOSYMBOL 0 0 8 0 0)
   (10 TypeSy $ADDRESS 0 0 BasicType 1)
   (11 TypeSy OBJECT 0 0 ClassType
      ()
   8 1 1 0)
   (12 ConstSy NIL 0 0 11 1 0)
   (13 ConstSy NULL 0 0 10 1 0)
   (14 ProcedureSy $MAIN 9 0
      ()
      (15)
   1 0)
   (15 VariableSy X 2 0 5 1 0)
   (16 TypeSy A 3 0 ArrayType 10 3 10)
   (17 TypeSy R 4 0 RecordType
      (18)
   1)
   (18 FieldSy x 4 0 1 1 0 17)
   (19 ConstSy C 5 0 1 1 10)
   (20 ProcedureSy P 7 0
      (21 22)
      ()
   0 2)
   (21 FormalSy x 6 1 2 1 9 0 1 VAR)
   (22 FormalSy y 6 1 17 1 9 1 2 VAL)
)
```

Note that this representation of the symbol table allows forward references. For example, symbol 17 (the record type `R`) is given before the declaration of the field 18 which it references.

B.2 A Final Example

```
PROGRAM P;

PROCEDURE Q (
    R :INTEGER);
BEGIN
  R := 5;
END;
VAR X : INTEGER;
VAR Y : REAL;
BEGIN
  IF X > 0 THEN
    Y := 5.5;
  ELSE
    WHILE X = 5 DO
      Y := Y + 1.0;
      Q(5);
      IF Y = 5.5 THEN
        X := 6;
      ENDIF;
    ENDDO;
  ENDIF;
END.
```

```
(
  (
    (1 TypeSy INTEGER 0 0 BasicType 1)
    (2 TypeSy REAL 0 0 BasicType 1)
    (3 TypeSy CHAR 0 0 BasicType 1)
    (4 TypeSy STRING 0 0 BasicType 0)
    (5 TypeSy BOOLEAN 0 0 EnumType
        (6 7)
    1)
    (6 EnumSy TRUE 0 0 5 0 1)
    (7 EnumSy FALSE 0 0 5 0 0)
    (8 TypeSy $NOTYPE 0 0 BasicType 0)
    (9 TempSy $NOSYMBOL 0 0 8 0 0)
    (10 TypeSy $ADDRESS 0 0 BasicType 1)
    (11 TypeSy OBJECT 0 0 ClassType
        ()
    8 1 1 0)
    (12 ConstSy NIL 0 0 11 1 0)
    (13 ConstSy NULL 0 0 10 1 0)
    (14 ProcedureSy $MAIN 21 0
        ()
        (17 18)
    2 0)
    (15 ProcedureSy Q 6 0
        (16)
        ()
    0 1)
    (16 FormalSy R 3 1 1 1 9 0 1 VAL)
    (17 VariableSy X 7 0 1 1 0)
    (18 VariableSy Y 8 0 2 1 1)
  )
```

```
  (
    (info 21 8 0 39 2 14 18)
    (begin 6 15 1 0 9 1 0 Q)
    (apush 5 16 R)
    (ipush 5 5)
    (istore 5)
    (end 6 15 Q)
    (begin 21 14 0 2 9 0 2 $MAIN)
    (apush 10 17 X)
    (iload 10)
    (ipush 10 0)
    (igt 10 2)
    (jmp 10 5)
    (apush 11 18 Y)
    (rpush 11 5.5)
    (rstore 11)
    (jmp 20 23)
    (apush 13 17 X)
    (iload 13)
    (ipush 13 5)
    (ieq 13 2)
    (jmp 13 18)
    (apush 14 18 Y)
    (apush 14 18 Y)
    (rload 14)
    (rpush 14 1.0)
    (radd 14)
    (rstore 14)
    (ipush 15 5)
    (call 15 15 Q)
    (apush 16 18 Y)
    (rload 16)
    (rpush 16 5.5)
    (req 16 2)
    (jmp 16 4)
    (apush 17 17 X)
    (ipush 17 6)
    (istore 17)
    (jmp 19 −21)
    (end 21 14 $MAIN)
  )
)
```