1 Introduction

Your task is to write an garbage collector for a small MODULA-2-like language called LUCA.

1. The garbage collector should be implemented using copying collection.
2. You should write your interpreter in C or C++ using gcc.
3. The interpreter should be named lucax. It should read the virtual machine code (in an S-expression format) from standard input.
4. You should test the interpreter on lectura.
5. You should work in a team of two students.
6. If you think it’s easier you can implement your interpreter using switch threading instead of indirect threading.
7. We won’t time your code but you should avoid writing your garbage collector 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 a LUCA program list.luc that creates a list of integers and prints it out:

PROGRAM list;

TYPE T = REF R;
TYPE R = RECORD[a:INTEGER; next:T];
VAR first : T;
VAR last : T;
VAR x : T;
VAR i : INTEGER;
BEGIN
  first := NEW T;
  first^.a := 0;
  first^.next := NULL;
  last := first;
FOR i := 1 TO 5 DO
  x := NEW T;
  x^.a := i;
  x^.next := NULL;
  last^.next := x;
  last := x;
ENDFOR;

x := first^.next;
WHILE x # NULL DO
  WRITE x^.a;
  Writeln;
  x := x^.next;
ENDDO;
END.

The complete syntax is given in Appendix A.

3 The Interpreter

Your interpreter only needs to support

- INTEGER, records, array, and reference (pointer) types.
- Assignment statements and the IF, WHILE, REPEAT, LOOP, EXIT, and FOR control statements.
- The built-in function NEW.
- The new statement GC which triggers a garbage collection.
- Integer constants and the built-in constant NULL.
- Integer and boolean expressions as well as array indexing, field and pointer dereferencing operators ([], ., and ^).

Your program should take two optional arguments:

-h size: Set the total heap size to size. size is in words.
-t Trace all heap operations. For every NEW operation print (to stderr)

NEW: allocated X bytes for type T.

Every time a garbage collection is triggered print

GC: START USED=... FREE=...
GC: END USED=... FREE=... WALL=... CPU=...

where FREE and USED is the amount of heap memory (in bytes) available and used and where WALL and CPU is the time (in seconds) used by the garbage collector as computed by this code:
#include<sys/resource.h>
#include<sys/time.h>
double GetTime () {
    struct timeval Time;
    double cpu, wall;
    struct rusage Resources;

    getrusage(RUSAGE_SELF, &Resources);
    Time = Resources.ru_utime;
    cpu = (double)Time.tv_sec +
        (double)Time.tv_usec/1000000.0;

    gettimeofday(&Time, NULL);
    wall = (double)Time.tv_sec +
        (double)Time.tv_usec/1000000.0;
}

Note:

- Don’t worry too much about the efficiency of your interpreter. Do worry about the efficiency of the garbage collector.
- You should trigger a garbage collection whenever a NEW is called and the heap has not enough free memory to honor the request.
- You should also perform a collection whenever the gc bytecode is executed.
- Don’t grow the heap.
- Generate an “out of memory error” when a NEW triggers a garbage collection and not enough memory is reclaimed to satisfy the request.
- The default heap size shall be 100 words.
- Pointers and INTEGERs are 64-bit quantities, i.e. 1 lectura word.

4 Virtual Machine Code

From this LUCA program

PROGRAM min;
TYPE T = REF INTEGER;
VAR x : T;
BEGIN
    x := NEW T;
    WRITE x^;
END.

the LUCA compiler generates this virtual machine code:

( (
Appendix B 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

1. instructions for dereferencing pointers and accessing record (struct) fields (refof, fieldof),
2. instructions for comparing pointers (ane, aeq),
3. the pushnull instruction which pushes the NULL value (0) on the stack, and
4. the new instruction which allocates an object from the heap.

You are, however, this time going to have to traverse and decode the symbol table in order to build templates. Keep in mind that data structures can be arbitrarily deeply nested: you can have arrays of arrays of records with fields that are pointers to arrays, etc. Thus, you need to examine the entries TypeSy, ArrayType, RefType, and RecordType.

So, what, do you ask, should I do???? Well, as usual, it’s always best to ![think](https://www.mediafire.com/...)

Here’s one way to go about working this assignment:

1. Implement pushnull, aeq, ane, refof, and new. For right now, make new just call malloc. Try this program:

```
PROGRAM min;
TYPE T = REF INTEGER;
VAR x : T;
BEGIN
```
x := NEW T;
WRITE x.a;
END.

2. Now implement fieldof. This instruction really is just an addition of the field offset (which you get from the symbol table) to the address of the record variable (which is on the stack)! Try this:

PROGRAM rec;
TYPE T = RECORD[a:INTEGER;b:INTEGER];
VAR x : T;
BEGIN
  WRITE x.a;
END.

3. Now, since you already have arrays implemented from the last assignment, this should just work:

PROGRAM long;
TYPE P = REF R;
TYPE R = RECORD[a:INTEGER;b:INTEGER];
TYPE T = ARRAY 100 OF P;
VAR x : T;
BEGIN
  x[4] := NEW P;
  x[4]^ .a := 42;
END.

Try some more complex examples, just to make sure. You don’t want interpreter errors to affect the garbage collector later on!

4. Create a special template for the global variables. This is simply a structure that holds a list of all the offsets of global variables which are pointers.

5. OK, time for the garbage collector itself! Parse the size of the heap from the command line, call malloc to create the heap, change new so that it gets a chunk of data off the new heap. Note that new should return an object with a template pointer as its first word. For example, if x := NEW T allocates on object of size 10 words, then you should malloc 11 words (8 * 11 = 88 bytes) and return a pointer to the 8:th byte. I.e., the pointer will be at offset x-8. Try a few small programs. Debug!

6. Now implement the copying collector! Split the heap in two, learn to follow pointers using the templates, do the copying when new runs out of memory. Start by trying to handle this little program:

PROGRAM glob;
TYPE T = REF INTEGER;
VAR x : T;
VAR y : T;
BEGIN
  x := NEW T;
  y := NEW T;
  x := NULL;
END.

After a collection the object that y points to should have been copied over.

7. Now is a good time to implement a print_heap function to be used during debugging!
8. Now it’s time to start building templates for the structured types! It would be possible to just let the S-expression symbol table entries be your templates, but that would be hard to walk during execution and inefficient also. Recursively walk the symbol table, extract the information you need (the kind of each type (record, array), the size of each type, and the offsets of record fields), and build the templates. You can simply store them in an array indexed by symbol table number. As a start, just handle non-recursive record types:

```plaintext
PROGRAM list;
TYPE T = REF R;
TYPE R = RECORD[a:INTEGER; next:T];
BEGIN
END.
```

Update the garbage collector to follow pointers from inside records!

9. Now handle arrays of pointers!

10. Try more complex types! Can you handle arrays of records of pointers? Records of arrays of pointers to records? Cover all the cases!

11. Lucky (for you) (a) you don’t have to handle procedures, and (b) through a quirk of syntax, the second statement isn’t legal LUCA:

```plaintext
PROGRAM bad;
TYPE P = REF INTEGER;
TYPE T = ARRAY 100 OF P;
VAR x : T;

PROCEDURE Q(t : P);
BEGIN
END;
BEGIN
Q(NEW P);
   x[(NEW P)\$] := NEW P;
END.
```

Why is this lucky? Well, it means that you don’t have to consider the stack as a source of roots when you start your garbage collection phase!

12. Try some real programs, debug and submit! 😊

6 Submission and Assessment

The deadline for this assignment is noon, Mon Mar 3. It is worth 10% of your final grade.

You should submit the assignment electronically using the Unix command

```bash
turnin cs520.2 lucax.c sexpr.h sexpr.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 | 
  'PROCEDURE' ident '(' [formal_list] ')' decl_list block '.

field ::= ident ':' ident ;

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
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:

<table>
<thead>
<tr>
<th>Left</th>
<th>Operators</th>
<th>Right</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>'+', '-', '*', '/', '%'</td>
<td>Int</td>
<td>Int</td>
</tr>
<tr>
<td>Real</td>
<td>'+', '-', '*', '/'</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Int</td>
<td>'&lt;', '&lt;=', '==', '#', '&gt;=', '&gt;'</td>
<td>Int</td>
<td>Bool</td>
</tr>
<tr>
<td>Real</td>
<td>'&lt;', '&lt;=', '==', '#', '&gt;=', '&gt;'</td>
<td>Real</td>
<td>Bool</td>
</tr>
<tr>
<td>Char</td>
<td>'&lt;', '&lt;=', '==', '#', '&gt;=', '&gt;'</td>
<td>Char</td>
<td>Bool</td>
</tr>
<tr>
<td>Bool</td>
<td>'AND', 'OR'</td>
<td>Bool</td>
<td>Bool</td>
</tr>
<tr>
<td></td>
<td>'NOT'</td>
<td>Bool</td>
<td>Bool</td>
</tr>
<tr>
<td></td>
<td>'-'</td>
<td>Int</td>
<td>Int</td>
</tr>
<tr>
<td></td>
<td>'TRUNC'</td>
<td>Real</td>
<td>Int</td>
</tr>
<tr>
<td></td>
<td>'FLOAT'</td>
<td>Int</td>
<td>Real</td>
</tr>
</tbody>
</table>

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 \[R := I \times R\] 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 \[\text{ARRAY 100 OF INTEGER}\] 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(info Pos Major Minor Instrs Globals Main Symbols)</td>
<td>[] ⇒ []</td>
</tr>
</tbody>
</table>

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.

| (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.

<table>
<thead>
<tr>
<th>(end Pos SyNo Name)</th>
<th>[] ⇒ []</th>
</tr>
</thead>
</table>

The end of the procedure.

<table>
<thead>
<tr>
<th>(call Pos ProcNo Name)</th>
<th>arg₁, arg₂,... ⇒ []</th>
</tr>
</thead>
</table>

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 its 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.
### B.1.2 Arrays, Records, and Pointers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(indexof Pos ArrayNo Name)</td>
<td>( [A, I] \rightarrow [A + I \times \text{ElmtSz}] )</td>
</tr>
</tbody>
</table>

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 \( I \times \text{ElmtSz} \), where \( \text{ElmtSz} \) 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fieldof Pos FieldNo Name)</td>
<td>( [R] \rightarrow [R + \text{FieldOffset}] )</td>
</tr>
</tbody>
</table>

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 \( \text{FieldOffset} \), the offset of the field within the record.

**FieldNo:** The symbol number of the field.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(refof Pos RefNo)</td>
<td>( [R] \rightarrow [R^\dagger] )</td>
</tr>
</tbody>
</table>

A address is on top of the stack. Push the value stored at that address.

**RefNo:** The symbol number of the pointer.

### B.1.3 Loading and Storing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ipush Pos Val)</td>
<td>( [] \rightarrow [\text{Val}] )</td>
</tr>
</tbody>
</table>

Push a literal value. \( \text{ipush} \) pushes an integer value, \( \text{rpush} \) a real, \( \text{cpush} \) a character, and \( \text{spush} \) a string. Pushing a string means pushing its address. For other types, the value of the constant is pushed.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pushnull Pos)</td>
<td>( [] \rightarrow [0] )</td>
</tr>
</tbody>
</table>

Push a NULL, i.e. 0, on the stack.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(apush Pos SyNo Name)</td>
<td>( [] \rightarrow [\text{addr(SyNo)}] )</td>
</tr>
</tbody>
</table>

Push the address of the local or global variable or formal parameter whose symbol number is \( \text{SyNo} \).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(tstore Pos )</td>
<td>( [L, R] \rightarrow [\ ] )</td>
</tr>
</tbody>
</table>

Store value \( R \) at address \( L \). \( t \in \{i, r, c\} \).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(tload Pos )</td>
<td>( [L] \rightarrow [R] )</td>
</tr>
</tbody>
</table>

Push the value \( R \) stored at address \( L \) onto the stack. \( t \in \{i, r, c\} \).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(astore Pos SyNo)</td>
<td>( [L, R] \rightarrow [\ ] )</td>
</tr>
</tbody>
</table>

Store value \( R \) (a pointer) at address \( L \). \( \text{SyNo} \) is the type of the pointer stored.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(copy Pos Type Size)</td>
<td>( [L, R] \rightarrow [\ ] )</td>
</tr>
</tbody>
</table>

Copy \( \text{Size} \) 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(new Pos SyNo)</td>
<td>( [\ ] \rightarrow [P] )</td>
</tr>
</tbody>
</table>

\( \text{SyNo} \) 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. The memory should be filled with zeroes.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(gc Pos SyNo)</td>
<td>( [\ ] \rightarrow [P] )</td>
</tr>
</tbody>
</table>

Perform a garbage collection.
### B.1.4 Arithmetic

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tadd Pos</code></td>
<td>([L,R] \Rightarrow [L + R])</td>
</tr>
<tr>
<td>Pop two elements off the stack and push their sum. (t \in {i,r}).</td>
<td></td>
</tr>
<tr>
<td><code>sub Pos</code></td>
<td>([L,R] \Rightarrow [L - R])</td>
</tr>
<tr>
<td>Pop two elements off the stack and push their difference. (t \in {i,r}).</td>
<td></td>
</tr>
<tr>
<td><code>mul Pos</code></td>
<td>([L,R] \Rightarrow [L * R])</td>
</tr>
<tr>
<td>Pop two elements off the stack and push their product. (t \in {i,r}).</td>
<td></td>
</tr>
<tr>
<td><code>div Pos</code></td>
<td>([L,R] \Rightarrow [L/R])</td>
</tr>
<tr>
<td>Pop two elements off the stack and push their quotient. (t \in {i,r}).</td>
<td></td>
</tr>
<tr>
<td><code>imod Pos</code></td>
<td>([L,R] \Rightarrow [L % R])</td>
</tr>
<tr>
<td>Pop two integers off the stack and push their remainder. I.e. there’s no floating point modulus operator.</td>
<td></td>
</tr>
<tr>
<td><code>uminus Pos</code></td>
<td>([L] \Rightarrow [-L])</td>
</tr>
<tr>
<td>Pop one element (L) off the stack and push (-L). (t \in {i,r}).</td>
<td></td>
</tr>
<tr>
<td><code>trunc Pos</code></td>
<td>([L] \Rightarrow \lfloor L \rfloor)</td>
</tr>
<tr>
<td>Convert (L) (a real) to an integer by rounding down.</td>
<td></td>
</tr>
<tr>
<td><code>float Pos</code></td>
<td>([L] \Rightarrow \text{float}(L))</td>
</tr>
<tr>
<td>Convert (L) (a integer) to the closest corresponding real.</td>
<td></td>
</tr>
</tbody>
</table>

### B.1.5 Input and Output

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>write Pos</code></td>
<td>([L] \Rightarrow [\ ])</td>
</tr>
<tr>
<td>Write (L) to standard output. (t \in {i,r,c,s}).</td>
<td></td>
</tr>
<tr>
<td><code>writeln Pos</code></td>
<td>([\ ] \Rightarrow [\ ])</td>
</tr>
<tr>
<td>Write a newline character.</td>
<td></td>
</tr>
<tr>
<td><code>read Pos Type</code></td>
<td>([L] \Rightarrow [\ ])</td>
</tr>
<tr>
<td>Read a value from standard input and store at address (L). (t \in {i,r,c}). It’s a fatal error if the value read is the wrong type.</td>
<td></td>
</tr>
</tbody>
</table>

### B.1.6 Branching

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>op Pos Offset</code></td>
<td>([L,R] \Rightarrow [\ ])</td>
</tr>
<tr>
<td>If (L \ op R) then goto (PC+\text{Offset}), where (PC) is the number of the current instruction. (t \in {i,r,c}), (op \in {\text{eq,ne,lt,gt,le,ge}}), i.e. there are instructions (\text{ieq, req, ceq, ine,rne, etc.})</td>
<td></td>
</tr>
<tr>
<td><code>aop Pos Offset</code></td>
<td>([L,R] \Rightarrow [\ ])</td>
</tr>
<tr>
<td>If (L \ op R) then goto (PC+\text{Offset}), where (PC) is the number of the current instruction. (op \in {\text{eq,ne}}). I.e. the two instruction (\text{aeq}) and (\text{ane}) compare addresses.</td>
<td></td>
</tr>
<tr>
<td><code>jmp Pos Offset</code></td>
<td>([\ ] \Rightarrow [\ ])</td>
</tr>
<tr>
<td>Goto (PC+\text{Offset}).</td>
<td></td>
</tr>
</tbody>
</table>

### B.1.7 Symbol Tables

The first part of the lvm code is always a symbol table. The format of each symbol is
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 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 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:
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_sems` for a simple program:

```bash
> 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 21 0 ...)
```
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.
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
    Q(5);
    IF Y = 5.5 THEN
      X := 6;
    ENDIF;
  ENDDO;
ENDIF;
END.

( ( 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 )
  ( ieq 10 2 )
  ( jmp 10 5 )
  ( apush 11 18 Y )
  ( rpush 11 5.5 )
  ( rstore 11 )
  ( iload 13 )
  ( jmp 20 23 )
  ( apush 13 17 X )
  ( iload 13 )
  ( ipush 13 5 )
  ( ieq 13 2 )
  ( jmp 13 18 )
  ( apush 14 18 Y )
  ( rpush 14 18 Y )
  ( iload 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 )
  ( ireq 16 2 )
  ( jmp 16 4 )
  ( apush 17 17 X )
  ( ipush 17 6 )
  ( istore 17 )
  ( jmp 19 -21 )
  ( end 21 14 $MAIN ) )
)