University of Arizona, Department of Computer Science

CSc 520 — Assignment 4 — Due noon, Wed, Apr 6 — 15%

Christian Collberg

March 23, 2005

1 Introduction

Your task is to write an interpreter and a garbage collector for a small MODULA-2-like language called LUCA. You can choose between *mark-and-sweep*, *copying collection*, and *generational collection*.

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/520/2005/Assignments`.

This assignment can be done individually or in a team of 2 students.

Make sure to hand in a README file describing, in detail, your implementation.

For this assignment efficiency of the garbage collector counts!

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 Luca language supports many data types (including arrays, records, classes and reals) and control structures and procedures/methods. Your interpreter, however, only needs to support

- INTEGER, BOOLEAN, records, and reference (pointer) types.
- Assignment statements and the IF, WHILE, REPEAT, LOOP, EXIT, and FOR control statements.
- The built-in function NEW.
- Integer constants and the built-in constant `NULL`.
- Integer and boolean expressions as well as the field and pointer dereferencing operators (`.` and `^`).

This will be enough to construct elaborate heap-structures that our garbage collector can work on. I.e. there will be no need for you to handle procedures, classes, reals, and arrays.

A complete grammar of Luca is given in Appendix A.

3 The Interpreter

A Luca compiler that generates stack virtual machine code will be given to you. You should write a C-program `lexec.c` that is called like this:

```
$ luca_lex list.luc | luca_parse | luca_sem -agvm | luca_AST2tree -agvm | \
        luca_tree2vm > list.vm
$ lexec list.vm
1
2
3
4
5
```

The first command compiles the program `list.luc` into a virtual machine code `list.vm`. The second runs your interpreter on this virtual machine code. All of the phases of the compiler pipeline read and generate S-expressions. The `-h` options lists other arguments to the different phases.

Your program should take two optional arguments:

-h *size*: Set the *total* heap size to *size.*

-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.
- The compiler (in particular the parser) is flaky.
- You should trigger a garbage collection whenever a NEW is called and the heap has not enough free memory to honor the request.
- Don't grow the heap.
- The default heap size shall be 1M bytes.
- Pointers and INTEGERs are 32-bit quantities.
- The virtual machine code is word addressed.
- We will test the interpreter on a Linux/x86 system.

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:

```
(
   (
      (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 0 0 0)
      (12 ConstSy NIL 0 0 11 0 0)
      (13 ConstSy NULL 0 0 10 0 0)
      (14 ProcedureSy $MAIN 8 0
         ()
         (16)
      0 0)
      (15 TypeSy T 3 0 RefType 1 0)
      (16 VariableSy x 4 0 15 0 0)
   )
   (
      (Info 8 7 8 10 0 14 16)
      (ProcBegin 8 14 0 1 9 0 0 $MAIN)
      (PushAddr 6 16 x)
      (NEW 6 15)
      (Store 6 15)
      (PushAddr 7 16 x)
      (RefOf 7 15)
      (Load 7 1)
```

```
        (Write 7 1)
        (ProcEnd 8 14 $MAIN)
    )
)
```

The vm code is in an S-expression format. The first part is the symbol table, the second part the code section.

5 Submission and Assessment

The deadline for this assignment is noon, Wed, Apr 6. You should submit the assignment (a text-file containing the function definitions) electronically using the `Unix` command ⌜`turnin cs520.4 lexec.c README`⌝. This assignment is worth 15% of your final grade.

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 TA.

A The Luca Syntax

Luca has constant and variable declarations, integer arithmetic, assignment statements, READ, WRITE, and WRITELN statements. Only integers and characters can be read, strings can also be written. Identifiers have to be declared before they are used. Identifiers cannot be redeclared. There are three (incompatible) built-in types, `INTEGER`, `BOOLEAN` and `CHAR`. The identifiers `TRUE` and `FALSE` are predeclared in the language.

`INTEGERS` and `REALS` are 32-bit quantities.

⟨*program*⟩ ::= '**PROGRAM**' ⟨*ident*⟩ ';' ⟨*decl_list*⟩ ⟨*block*⟩ '.'
⟨*block*⟩ ::= '**BEGIN**' ⟨*stat_seq*⟩ '**END**'
⟨*decl_list*⟩ ::= { ⟨*declaration*⟩ ';' }
⟨*declaration*⟩ ::= '**CONST**' ⟨*ident*⟩ ':' ⟨*ident*⟩ '=' ⟨*expression*⟩
    '**VAR**' ⟨*ident*⟩ ':' ⟨*ident*⟩
⟨*expression*⟩ ::= ⟨*expression*⟩ ⟨*bin_operator*⟩ ⟨*expression*⟩ |
    ⟨*unary_operator*⟩ ⟨*expression*⟩ |
    '(' ⟨*expression*⟩ ')' | ⟨*integer_literal*⟩ | ⟨*char_literal*⟩ | ⟨*designator*⟩
⟨*designator*⟩ ::= ⟨*ident*⟩
⟨*bin_operator*⟩ ::= '+' | '−' | '∗' | '/' | '%'
⟨*unary_operator*⟩ ::= '−'
⟨*stat_seq*⟩ ::= { ⟨*statement*⟩ ';' }
⟨*statement*⟩ ::= ⟨*designator*⟩ ':=' ⟨*expression*⟩
    '**WRITE**' ⟨*expression*⟩ | '**WRITELN**' |
    '**READ**' ⟨*designator*⟩

Luca has IF, IF-ELSE, LOOP, EXIT, REPEAT, FOR, and WHILE statements. EXIT statements can only occur within LOOP statements. The expression in an IF, IF-ELSE, REPEAT, or WHILE statement must be of boolean type.

$\langle$*bin_operator*$\rangle$ ::= '**AND**' | '**OR**' | '<' | '<=' | '=' |'#' | '>=' |'>'

$\langle$*unary_operator*$\rangle$ ::= '**NOT**'

$\langle$*statement*$\rangle$ ::= '**IF**' $\langle$*expression*$\rangle$ '**THEN**' $\langle$*stat_seq*$\rangle$ '**ENDIF**'|
'**IF**' $\langle$*expression*$\rangle$ '**THEN**' $\langle$*stat_seq*$\rangle$ '**ELSE**' $\langle$*stat_seq*$\rangle$ '**ENDIF**' |
'**WHILE**' $\langle$*expression*$\rangle$ '**DO**' $\langle$*stat_seq*$\rangle$ '**ENDDO**' |
'**REPEAT**' $\langle$*stat_seq*$\rangle$ '**UNTIL**' $\langle$*expression*$\rangle$ |
'**LOOP**' $\langle$*stat_seq*$\rangle$ '**ENDLOOP**' |
'**EXIT**' |
'**FOR**' $\langle$*ident*$\rangle$ ':=' $\langle$*expression*$\rangle$ '**TO**' $\langle$*expression*$\rangle$ ['**BY**' $\langle$*const_expr*$\rangle$] '**DO**' $\langle$*stat_seq*$\rangle$ '**ENDFOR**'

The **FOR**-loop **BY**-expression must be a compile-time constant expression. A Luca `FOR`-loop

```
FOR i := e1 TO e2 BY e3 DO
   S
ENDFOR
```

is compiled into code that's equivalent to

```
i := e1;
T1 := e2;
T2 := e3;
IF T2 >= 0 THEN
   WHILE i <= T1 DO
      S;
      i := i + T2;
   ENDDO;
ELSE
   WHILE i >= T1 DO
      S;
      i := i + T2;
   ENDDO;
ENDIF
```

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.

We add two operators TRUNC and FLOAT:

$\langle$*unary_operator*$\rangle$ ::= '**TRUNC**' | '**FLOAT**'

Luca has arrays, record, and pointer types.

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).

The element count of an array declaration must be a constant integer expression. 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.

Here are the extensions to the concrete syntax:

$\langle$*declaration*$\rangle$ ::= '**TYPE**' $\langle$*ident*$\rangle$ '=' '**ARRAY**' $\langle$*expression*$\rangle$ '**OF**' $\langle$*ident*$\rangle$
'**TYPE**' $\langle$*ident*$\rangle$ '=' '**RECORD**' '[' { $\langle$*field*$\rangle$ } ']'
'**TYPE**' $\langle$*ident*$\rangle$ '=' '**REF**' $\langle$*ident*$\rangle$
$\langle$*field*$\rangle$ ::= $\langle$*ident*$\rangle$ ':' $\langle$*ident*$\rangle$ ';'
$\langle$*designator*$\rangle$ ::= $\langle$*ident*$\rangle$ { $\langle$*designator'*$\rangle$ }
$\langle$*designator'*$\rangle$ ::= '[' $\langle$*expression*$\rangle$ ']' $\langle$*designator'*$\rangle$ | '.' $\langle$*ident*$\rangle$ $\langle$*designator'*$\rangle$ | '^' $\langle$*designator'*$\rangle$
$\langle$*unary_operator*$\rangle$ ::= '**NEW**' $\langle$*ident*$\rangle$

The NEW operator takes a reference type as parameter and returns a new dynamic variable as result. The ^ operator dereferences a pointer.

Luca has non-nested procedures. There is no limit to the number of arguments a procedure may take. Value parameters (including structured types such as arrays and records!) should be passed by value, VAR parameters by reference. Procedures can be recursive.

$\langle$*declaration*$\rangle$ ::= '**PROCEDURE**' $\langle$*ident*$\rangle$ '(' [$\langle$*formal_list*$\rangle$] ')' $\langle$*decl_list*$\rangle$ $\langle$*block*$\rangle$ ';'
$\langle$*formal_list*$\rangle$ ::= $\langle$*formal_param*$\rangle$ { ';' $\langle$*formal_param*$\rangle$ }
$\langle$*formal_param*$\rangle$ ::= ['**VAR**'] $\langle$*ident*$\rangle$ ':' $\langle$*ident*$\rangle$
$\langle$*actual_list*$\rangle$ ::= $\langle$*expression*$\rangle$ { ',' $\langle$*expression*$\rangle$ }
$\langle$*statement*$\rangle$ ::= $\langle$*ident*$\rangle$ '(' [$\langle$*actual_list*$\rangle$] ')'
$\langle$*field*$\rangle$ ::= $\langle$*ident*$\rangle$ ':' $\langle$*ident*$\rangle$ ';'

B Virtual Machine Code

Here is a description of the Luca virtual machine, lvm. Note that some of this information does not pertain to this assignment, since you are not required to handle procedures, arrays, etc.

- lvm is a word-addressed machine. Words are 32 bits wide. The size of all basic types (integers, reals, booleans, and chars) is one word.

- 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.

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.	$[\,] \Rightarrow [\,]$
(**ProcBegin** `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.	$[\,] \Rightarrow [\,]$
(**ProcEnd** `Pos SyNo Name`) The end of the procedure.	$[\,] \Rightarrow [\,]$

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 $I *$ `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 Type)` Push the value R stored at address L onto the stack. Note that this is essentially equivalent of the `Load` instruction, but generated only from the pointer dereferencing operator.	$[L] \Rightarrow [R]$
(PushAddr `Pos SyNo Name)` Push the address of the local or global variable or formal parameter whose symbol number is `SyNo`.	$[\ ] \Rightarrow [\texttt{addr(SyNo)}]$
(PushImm `Pos Type Val)` Push a literal value. Pushing a string means pushing its address. For other types, the value of the constant is pushed.	$[\ ] \Rightarrow$ [Val]
(Store `Pos Type)` Store value R at address L.	$[L, R] \Rightarrow [\ ]$
(Load `Pos Type)` Push the value R stored at address L onto the stack.	$[L] \Rightarrow [R]$
(Copy `Pos Type Size)` 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.	$[L, R] \Rightarrow [\ ]$
(BinExpr `Pos Op Type)` Integer or real arithmetic. `Type` is the symbol number of the integer or real standard type.	$[L,R] \Rightarrow [L\ \texttt{Op}\ R]$
(UnaryExpr `Pos Op Type)` Integer or real unary arithmetic. `Op` is `TRUNC`, `FLOAT`, or `-` (unary minus).	$[L,R] \Rightarrow [\texttt{Op}\ L]$
(Write `Pos Type)` Write L.	$[L] \Rightarrow [\ ]$
(NEW `Pos Type)` Generate a new dynamic variable of type `Type` and push its address on the stack.	$[] \Rightarrow [R]$

Instruction	Stack
(**PushNull** `Pos Type`) Push `NULL` (0) on the stack.	[] ⇒ [0]
(**WriteLn** `Pos`) Write a newline character.	[ ] ⇒ [ ]
(**Read** `Pos Type`) Read a value and store at address L.	[L] ⇒ [ ]
(**ProcCall** `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.	[ ] ⇒ [ ]
(**Branch** `Pos Op Type Offset`) If L `Op` R then goto `PC`+`Offset`, where `PC` is the number of the current instruction. Only integers, reals, characters, and booleans can be compared.	[L,R] ⇒ [ ]
(**Goto** `Pos Offset`) Goto `PC`+`Offset`.	[ ] ⇒ [ ]

B.1 The Symbol Tables

The format of each symbol is

(*number kind name position level* ...)

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. For the purposes of this assignment these can be set to 0.

(***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. `size` and `offset` can be set to 0. `copy` should be set to 9 (`$NOSYMBOL`). `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. `size` and `offset` can be set to 0. `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. `localSize` and `formalSize` can be set to 0.

(***number*** **TypeSy** ***name pos level*** **BasicType** `size`)
: This represents a basic type such as integer or real. `size` can be set to 0.

(***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. `size` can be set to 0.

(***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. `size` can be set to 0.

(***number*** **TypeSy** ***name pos level*** **RefType** ***referent*** `size`)
: This represents a reference (pointer) type. Where *referent* is the type pointed to.

(***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. `size` can be set to 0.

Some symbols are predeclared by the compiler:

```
    (1 TypeSy INTEGER 0 0 BasicType)
    (2 TypeSy REAL 0 0 BasicType)
    (3 TypeSy CHAR 0 0 BasicType)
    (4 TypeSy STRING 0 0 BasicType)
    (5 TypeSy BOOLEAN 0 0 EnumType)
    (6 EnumSy TRUE 0 0 5 0 1)
    (7 EnumSy FALSE 0 0 5 0 0)
    (8 TypeSy $NOTYPE 0 0 BasicType)
    (9 TempSy $NOSYMBOL 0 0 8 0 0)
    (10 TypeSy $ADDRESS 0 0 BasicType)
    (11 ProcedureSy $MAIN 7 0 () () 0 0)
```

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

```
> cat t.gus
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.
> luca_lex t.gus | luca_parse | luca_sem
(
   (
      (12 VariableSy X 2 0 5 1 0)
      (13 TypeSy A 3 0 ArrayType 10 3 10)
      (14 TypeSy R 4 0 RecordType (15) 4)
      (15 FieldSy x 4 0 1 4 0 14)
      (16 ConstSy C 5 0 1 4 10)
      (17 ProcedureSy P 7 0 (18 19) () 0 8)
      (18 FormalSy x 6 1 2 4 9 0 1 VAR)
      (19 FormalSy y 6 1 14 4 9 4 2 VAL)
   )
   (PROGRAM P 10 ... )
)
```

Note that this representation of the symbol table allows forward references. For example, symbol 14 (the record type R) is given before the declaration of the field 15 which it references. This is acceptable, but not required, behavior.