University of Arizona, Department of Computer Science

CSc 520 — Assignment 1 — Due noon, Mon Feb 11 — 10%

Christian Collberg
January 28, 2008

1 Introduction

Your task is to write an interpreter for a small subset of the language Luca. You will be given a front-end that performs lexing, parsing, semantic analysis, and intermediate code generation on Luca source files. You will write an interpreter that reads in the code produced by the front-end, and then executes this code.

1. The interpreter should be implemented using *indirect threaded code*.
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 only have to implement control structures (`IF`, `WHILE`, etc.), integer and real arithmetic, a `WRITE` statement, and array indexing.
5. You should test the interpreter on lectura.
6. You should work in a team of two students.
7. Future assignments will build on this one so it is in your best interest to do a good job! If you have plenty of time on your hand now at the beginning of the semester you can go ahead and implement procedures (which we'll need for the assignments on object oriented programming) and/or records and pointers (which we'll need when we implement a garbage collector). We won't test any of these features for this assignment, though.

The Luca front-end can be downloaded from `http://www.cs.arizona.edu/~collberg/Teaching/520/2008/Assignments`. The compiler/interpreter is invoked like this:

```
> lc quicksort.luc | lucax
```

If you're curious of how the Luca compiler works internally, you can look at the output from the various compiler phases:

```
> lc quicksort.luc -L lex
> lc quicksort.luc -L parse
> lc quicksort.luc -L sem
> lc quicksort.luc -L tree
> lc quicksort.luc -L stack
```

2 The Luca Language

Luca is similar to Modula-2 and Modula-3. The complete syntax is given in Appendix A. Below is Ackermann's function coded in Luca. However, you will only have to implement a small subset consisting of integer and real arithmetic, control structures, and arrays.

```
PROGRAM Ackermann;

PROCEDURE Pow(base : INTEGER; exp : INTEGER; VAR Out : INTEGER);
VAR i : INTEGER;
BEGIN
   Out := 1;
   FOR i := 1 TO exp DO
      Out := Out * base;
   ENDFOR;
END;

PROCEDURE A(i : INTEGER; j : INTEGER; VAR R : INTEGER);
VAR R1 : INTEGER;
BEGIN
   IF i = 1 AND j >= 1 THEN
      Pow(2, j, R);
   ELSE
       IF i >= 2 AND j = 1 THEN
          A(i-1, 2, R);
       ELSE
          IF i >= 2 AND j >= 2 THEN
              A(i, j-1, R1);
              A(i-1, R1, R);
          ENDIF;
       ENDIF;
   ENDIF;
END;

VAR R     : INTEGER;
BEGIN
   A(2, 3, R);
   WRITE R;
   WRITELN;
END.
```

In a Luca program the main program is between the last `BEGIN-END`. A formal **VAR**-parameter is passed by reference, i.e. its address is passed rather than its value.

3 The Bytecode

The front-end reads, parses, and performs semantic analysis on Luca programs. it also generates intermediate code; a typed stack code. Here's a simple Luca program:

```
PROGRAM P;
VAR X : INTEGER;
BEGIN
    X := 5;
    WRITE X;
END.
```

Here is the corresponding stack code:

```
(
   (
      ...
      (15 VariableSy X 2 0 1 1 0)
   )
   (
      (info 6 8 0 9 1 14 15)
      (begin 6 14 0 1 9 0 1 $MAIN)
      (apush 4 15 X)
      (ipush 4 5)
      (istore 4)
      (apush 5 15 X)
      (iload 5)
      (iwrite 5)
      (end 6 14 $MAIN)
   )
)
```

There are two major parts: the symbol table and a list of stack instructions.

The instructions of each procedure are numbered starting at 1. In the example above there is one instruction per line. The first word is the opcode, the second the source postion, then follows a possibly empty list of extra arguments.

Appendix B has a complete listing of all the bytecodes.

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

For this assignment you don't have to implement all of Luca! You don't have to implement procedures, records, and pointer types, for example. Rather, the only instructions you have to worry about are:

1. load and store instructions for integers and reals (`iload`, `rload`, `istore`, `rstore`),
2. arithmetic instructions for integers and reals (`iadd`, `radd`, ..., `iuminus`, `ruminus`, `trunc`, `float`),
3. comparison instructions for integers and reals (`ine`, `rne`, ...),
4. the unconditional branch (`jmp`),
5. the array indexing operator (`indexof`),
6. the instructions for pushing integer and real literal values (`ipush`, `rpush`),

7. `apush`, the instruction for pushing the address of a variable onto the stack, and
8. the output instructions for integers and reals `iwrite` and `rwrite`, and `writeln`.

The only part of the symbol table you need to read is `VariableSy` (to get the offset of each variable which the `apush` instruction pushes on the stack) and `ArraySy` (to get the length of the array so that you can implement bounds checks).

So, what, do you ask, should I do???? Well, 

, here's a schedule for you:

1. Start by downloading the compiler and try it out on some simple programs. Learn what each bytecode does by looking at the generated code and the description of the bytecode in Appendix B.
2. A C module `sexpr` will be given to you. It parses an S-expression (the representation that the intermediate code is stored in) into a data structure which you can then walk to get the information you need. Try it out:

```
#include <stdio.h>
#include "sexpr.h"

int main (int argc,char** argv) {
    SExpr* root;
    if (!parse(argv[1], &root)){
        printf("can't open file\n");
    }else{
        print_SExpr(root);
        free_SExpr(root);
    }
}
```

3. Continue playing with `sexpr`: learn how to walk the S-expression data-structure to get out each instruction and its arguments.
4. Assign bytecodes to each instruction. For example, let `iadd=1`, `isub=2`, etc. Read in each instruction from the S-expression, convert to the bytecode, and store the result in a bytecode array. Keep in mind that some instruction arguments are one word long, not one byte. This is true of the `ipush` and `rpush` instructions, for example, whose constant value argument is 32 bits long.
5. Code up an evaluation stack with `push` and `pop` functions. While you're at it, add a `print_stack` function for debugging.
6. Write a simple *switch*-based interpreter for a small subset of the instructions: `iadd`, `ipush`, `iload`, `istore`, `iwrite`, and `writeln`. Debug! See, that wasn't hard, was it? 
7. Now convert the *switch*-based interpreter to the *indirect threaded one*. Ooops! Segmentation fault???  Well, time for more debugging!
8. Add the remaining instructions. Debug your interpreter on larger programs. Cover all the corner cases.
9. Submit! 

Note that we will test your interpreter by running it on a set of micro-programs. For example, to test whether `+` works, we might run the program below:

```
PROGRAM intadd;
VAR x : INTEGER;
VAR y : INTEGER;
BEGIN
   x := 5;
   y := 66;
   WRITE x + y; WRITELN;
END.
```

Note that it is not possible to test each bytecode in complete isolation. The program above, for example, makes use of the bytecodes for load, store, and write and if these don't work it will appear as if `+` doesn't work either.

5 Submission and Assessment

The deadline for this assignment is noon, Mon Feb 11. It is worth 10% of your final grade.

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

```
turnin cs520.1 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 '=' '**RECORD**' '[' [field_list] ']' |
    '**TYPE**' ident '=' '**ARRAY**' expression '**OF**' ident |
    '**CONST**' ident ':' ident '=' expression |
    '**PROCEDURE**' ident '(' [formal_list] ')' decl_list block ';'

field_list ::=
    field_decl { ';' field_decl }

field_decl ::=
    ident ':' ident

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**' |
    '**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

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

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

- The LUCA language is case sensitive.
- Luca has four (incompatible) built-in types: `INTEGER`, `CHAR`, `BOOLEAN` and `REAL`. All basic types are 32 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.
- 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` and `FALSE` 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 32 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 4 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. `Pos` and `Name` should not be used by the interpreter — they are there for ease of debugging only.

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 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 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 Pos SyNo Name)` The end of the procedure.	$[\,] \Rightarrow [\,]$
(call `Pos 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.	$[\mathrm{arg}_1, \mathrm{arg}_2, \ldots] \Rightarrow [\,]$

B.1.2 Arrays, Records, and Pointers

Instruction	Stack
(indexof `Pos 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 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 Pos RefNo)` A address is on top of the stack. Push the value stored at that address. `RefNo`**:** The symbol number of the pointer.	$[R] \Rightarrow [R\hat{}\,]$

B.1.3 Loading and Storing

Instruction	Stack
(*t*push `Pos Pos 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.	$[\,] \Rightarrow [\mathrm{Val}]$
(apush `Pos Pos SyNo Name)` Push the address of the local or global variable or formal parameter whose symbol number is `SyNo`.	$[\,] \Rightarrow [\texttt{addr(SyNo)}]$
(*t*store `Pos Pos)` Store value R at address L. $t \in \{\texttt{i}, \texttt{r}, \texttt{c}\}$.	$[L, R] \Rightarrow [\,]$
(*t*load `Pos Pos)` Push the value R stored at address L onto the stack. $t \in \{\texttt{i}, \texttt{r}, \texttt{c}\}$.	$[L] \Rightarrow [R]$
(copy `Pos 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.	$[L, R] \Rightarrow [\,]$

B.1.4 Arithmetic

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

B.1.5 Input and Output

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

B.1.6 Branching

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

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. `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.

(***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*** **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)
  )
)
```