1 Introduction

Your task is to write a code generator for the language LUCA. You will be given a front-end that produces a control flow graph of

a) expression trees,

b) stack code,

c) quadruples, or

d) triples

from LUCA source files. You should write a code generator that produces either

a) SPARC-V9 assembly code that runs on lectura, or

b) MIPS assembly code that runs on the SPIM simulator

from one of these intermediate forms.

You should produce a program lc that performs all the necessary actions to turn a source program into assembly code. It should be called and executed like this (if you are compiling for the MIPS):

```
> lc x.gus > x.s
> spim -f x.s
```

or like this (if you are compiling for the SPARC):

```
> lc x.gus > x.s
> gcc x.s
> a.out
```

Typically, lc will be a shell script that calls the various pieces of the luca compiler suite:

```
#!/bin/sh
luca_lex $1 | luca_parse | luca_sem -amips | luca_AST2tree |\
luca_tree2quad -C | luca_mips
```
The exact details of this script of course depend on your particular implementation.

Note that if you are compiling for the MIPS, `luca_sem` must be run with the `-amips` option. If you are compiling for the SPARC, instead use the `-asparc-v9` option.

There are five different versions of LUCA described in this document. You can decide to write your code generator for any one of them. If you decide to work on LUCA-1, for example, the maximum number of points you can receive is 60; if you work on LUCA-4 you can receive 90 points, etc. It’s a good idea to start with LUCA-1 and then incrementally add new features until you are compiling your chosen target language.

- You may use whatever code generation algorithm you want. I will only grade on correctness, not the speed of the generated code.
- If you feel you’ve done something very clever I might be convinced to give extra credit. Be sure to indicate what you have implemented in your README file.
- This assignment can be coded in the language of your choice as long as it can be compiled and run on lectura.
- Make sure that your Makefile is working properly, and that `lc` is called exactly as in the example above.
- You can assume that the input generated by the front-end is correct. In other words, you don’t have to do any input error checking.
- Your code generator must generate code from a control-flow graph, not from the raw intermediate instructions. The LUCA intermediate code generators (`luca_AST2tree, luca_tree2*`) take an argument `-C` which will generate a CFG for you, but you may, if you wish, build one yourself.

Hint: There will be one more assignment where you will be asked to do code local optimization on the intermediate code and/or generated machine code. You will only have one week to complete this assignment. For this reason, it may be in your best interest to prepare for the next assignment at this time. In particular, rather than simply printing out each machine code instruction as you generate it, it is better to make each instruction a record/class/struct like this:

```java
class MipsInstruction {
    String label;
    String operator;
    Operand arg1;
    Operand arg2;
    Operand arg3;
}
abstract class Operand {
}
class Register extends Operand {
    String name;
}
class Immediate extends Operand {
    int value;
}
class RegPlusOffs extends Operand {
    String name;
    int value;
}
```
and to insert these instructions in a linked list. This will allow the code optimizer to run over the generated instructions and improve on them.

2 Luca-1 [60 Points]

Luca-1 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. Here is the concrete syntax of Luca-1:

\[
\begin{align*}
\text{program} & ::= \text{`PROGRAM'} \text{ (ident) '} \text{ (decl_list) (block) '} \\
\text{block} & ::= \text{`BEGIN'} \text{ (stat_seq) `END'} \\
\text{decl_list} & ::= \{ \text{ (declaration) '} \} \\
\text{declaration} & ::= \text{`CONST'} \text{ (ident) '} \text{ (ident) '=} \text{ (expression)} \\
\text{VAR} & ::= \text{ (ident) '} \text{ (ident)} \\
\text{expression} & ::= \text{ (expression) (bin_operator) (expression)} | \\
& \quad \text{ (unary_operator) (expression) } | \\
& \quad \text{ (designator)} \\
\text{designator} & ::= \text{ (ident)} \\
\text{bin_operator} & ::= \text{ `+' | `-' | `*' | `/'} | `%' \\
\text{unary_operator} & ::= `-' \\
\text{stat_seq} & ::= \{ \text{ (statement) '} \} \\
\text{statement} & ::= \text{ (designator) '=} \text{ (expression)} \\
\text{WRITE} & ::= \text{ (expression)} | \text{WRITELN} | \\
\text{READ} & ::= \text{ (designator)}
\end{align*}
\]

On the MIPS, integers are 32-bit quantities, character and booleans are 8-bits. On the SPARC, integers are 64-bit quantities, character and booleans are 8-bits.

On the SPARC, READ and WRITE should be implemented by calling functions in the standard C IO-library stdio. To see how to do this, inspect the output of the standard C compiler. For example, compile

```c
#include <stdio.h>
main (){printf("%i\n",56);}]
```

with `cc -S test.c`. The resulting assembly code contains the calling sequence for printf.

3 Luca-2 [10 Points]

Luca-2 adds 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. Here are the extensions to the concrete syntax:
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

should be 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

**Hint:** The front-end of the compiler takes care of generating the right intermediate code for **FOR**-loops, so there really isn’t anything special you have to do in the code generator to handle them.

4 **LUCA-3** [10 Points]

**LUCA-3** extends **LUCA-2** with real variable declarations and real arithmetic.

**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 \texttt{TRUNC} and \texttt{FLOAT}:

\[
\texttt{TRUNC} | \texttt{FLOAT}
\]

On the MIPS, reals are 32-bit quantities. On the SPARC, reals are 64-bit quantities.

\section{5 \textsc{Luca-4} [10 Points]}

\textsc{Luca-4} extends \textsc{Luca-3} with one-dimensional arrays and record types.

Assignment is defined for scalars only, not for variables of structured type. In other words, the assignment \texttt{A:=B} is illegal if \texttt{A} or \texttt{B} are records or arrays. \texttt{READ} and \texttt{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 \texttt{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:

\[
\texttt{TYPE} \langle \texttt{ident} \rangle \texttt{=} \texttt{ARRAY} \langle \texttt{expression} \rangle \texttt{OF} \langle \texttt{ident} \rangle
\]

\[
\texttt{TYPE} \langle \texttt{ident} \rangle \texttt{=} \texttt{RECORD} [\langle \texttt{field} \rangle ]
\]

\[
\langle \texttt{field} \rangle \texttt{=} \langle \texttt{ident} \rangle \texttt{;}\langle \texttt{ident} \rangle \texttt{;}
\]

\[
\langle \texttt{designator} \rangle \texttt{=} \langle \texttt{ident} \rangle \{ \langle \texttt{designator} \rangle \}
\]

\[
\langle \texttt{actual} \rangle \texttt{=} \langle \texttt{expression} \rangle [\langle \texttt{ident} \rangle \langle \texttt{designator} \rangle ]
\]

\[
\langle \texttt{statement} \rangle \texttt{=} \langle \texttt{ident} \rangle \langle \texttt{expression} \rangle \langle \langle \texttt{actual} \rangle \rangle
\]

\[
\langle \texttt{field} \rangle \texttt{=} \langle \texttt{ident} \rangle \langle \texttt{field} \rangle \langle \texttt{ident} \rangle
\]

\section{6 \textsc{Luca-5} [10 Points]}

\textsc{Luca-5} extends \textsc{Luca-4} with 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, \texttt{VAR} parameters by reference. Procedures can be recursive.

Here are the extensions to the concrete syntax:

\[
\texttt{PROCEDURE} \langle \texttt{ident} \rangle \langle \langle \texttt{formal} \rangle \rangle \langle \langle \texttt{actual} \rangle \rangle \langle \texttt{block} \rangle ;
\]

\[
\texttt{formal} \langle \texttt{list} \rangle \texttt{=} \langle \texttt{formal} \rangle \langle \langle \texttt{formal} \rangle \rangle \langle \texttt{decl} \rangle \langle \texttt{block} \rangle ;
\]

\[
\texttt{formal} \langle \texttt{param} \rangle \texttt{=} \langle \texttt{formal} \rangle \langle \langle \texttt{formal} \rangle \rangle \langle \texttt{var} \rangle \langle \texttt{ident} \rangle
\]

\[
\texttt{actual} \langle \texttt{list} \rangle \texttt{=} \langle \texttt{expression} \rangle \langle \langle \texttt{var} \rangle \rangle \langle \texttt{ident} \rangle
\]

\[
\texttt{statement} \texttt{=} \langle \texttt{ident} \rangle \langle \langle \texttt{expression} \rangle \rangle \langle \langle \texttt{actual} \rangle \rangle
\]

\[
\texttt{field} \texttt{=} \langle \texttt{ident} \rangle \langle \texttt{field} \rangle \langle \texttt{ident} \rangle
\]
7 Submission and Assessment

The deadline for this assignment is midnight, December 4. You should submit the assignment electronically using the Unix command `turnin cs453.5 <files> README ...`.

Your submission must contain a README-file that states which parts of LUCA your code generator 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 code generator. 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!

This assignment is worth 100 points.

Don’t show your code to anyone outside your team, 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 TA or the instructor.

A The tree intermediate code

The front-end generates control flow graphs where each node is a sequence of expression trees. The nodes in these trees are defined below.

### Declarations

- **(Version Pos Major Minor)** The version of the intermediate code language.
- **(VarDecl Pos Symbol)** VarDecl declares a global or local variable. Symbol is the symbol number.
- **(FormalDecl Pos Symbol)** Declares the formal parameter of a procedure. Symbol is the symbol number.
- **(TypeDecl Pos Symbol)** Declares a record or array type. Symbol is the symbol number.

### Loads and Stores

- **(Store Pos Type Left Right)** Left is an expression tree computing an address. Right is an expression tree computing a value (it’s type is given by symbol Type) to be stored at that address.
- **(Load Pos Type Des)** Des is an expression tree computing an address. Load should load the value (whose type is given by the symbol Type) stored at that address.
### Expressions

- **(BinExpr Pos Op Type Left Right)**: A node in an expression tree that computes \( \text{Left \ Op \ Right} \). \( \text{Op} \) is a string. \( \text{Type} \) is a symbol number.

- **(UnaryExpr Pos Op Type Left)**: A node in an expression tree that computes \( \text{Op \ Left} \). \( \text{Op} \) is a string. \( \text{Type} \) is a symbol number.

- **(LoadLit Pos Type Value)**: Load the literal value \( \text{Value} \). In case of strings, the address should be loaded, not the value.

### Designators

- **(AddressOf Symbol Type)**: Load the address of \( \text{Symbol} \) (which could be a global variable, local variable, or formal parameter). \( \text{Type} \) is the type of the symbol.

- **(IndexOf Pos Type Base Index)**: Compute the address of an array element, i.e. \( \text{Base} + \text{ElementType} \times \text{Index} \). \( \text{Base} \) is an expression tree computing the base address of the array. \( \text{Index} \) is an expression tree computing the index value. \( \text{Type} \) is the symbol number of the array. It’s a checked, fatal, run-time error for \( \text{Index} \) to be \(<0 \) or \( >\text{HighBound(Type)}-1 \).

- **(FieldOf Pos Type Field Base)**: Compute the address of a record field, i.e. \( \text{Base} + \text{Offset(Field)} \). \( \text{Base} \) is an expression tree computing the base address of the record. \( \text{Field} \) is the symbol number for the field from which we can retrieve information such as offset. \( \text{Type} \) is the symbol number of the record type.

### Control

- **(Branch Pos Op Type Left Right Label)**: Equivalent to if \( \text{Left \ Op \ Right} \) then goto \( \text{Label} \). \( \text{Left} \) and \( \text{Right} \) are expression trees, \( \text{Op} \) is a string. If we are generating control flow graphs \( \text{Label} \) is the number of the basic block to which we should jump. If we are generating a raw sequence of trees \( \text{Label} \) is the label number (from a \( \text{(Label Pos Number)} \) node) we should jump to.

- **(Goto Pos Label)**: Jump to basic block/instruction number \( \text{Label} \).

- **(Label Pos Label)**: Marks a position in the code to which a Branch or Goto instruction can jump. \( \text{Label} \) is an integer.

### Input and Output

- **(Write Pos Type Expr)**: Write the value of \( \text{Expr} \) (an expression tree) to the standard output. If \( \text{Expr} \) is a (constant) string, \( \text{Expr} \) will compute the string’s address, not its value.

- **(Read Pos Type Des)**: Read a value into the address held by \( \text{Des} \), an expression tree. The type of the data to be read is given by \( \text{Type} \), a symbol number.

- **record (WriteLn Pos)**: Write a newline character to the standard output.
B The control flow graph

The intermediate code instructions are organized in a series of control flow graphs, one per procedure.

Here is an example:

```plaintext
> cat 0.gus

PROGRAM P;
BEGIN
  IF 1.1 < 2.0 THEN
    WRITE 0;
  ENDIF;
END.
```

```plaintext
> luca_lex 0.gus | luca_parse | luca_sem | luca_AST2tree -C

( (11 ProcedureSy $MAIN 6 0
 ()
 ()
 0 0)
 )

( PROCEDURE 11 1 6
 (BLOCK 1  <<== The Entry block
 ()  <<== Predecessor blocks
 (2)  <<== Successor blocks
 ()  <<== List of instructions
 )
)
(BLOCK 2
(1)
(3 4)
(
(Branch 3 "<" 2 4
  (LoadLit 3 2 1.1)
  (LoadLit 3 2 2.0)
)
)
)
(BLOCK 3
(2)
(5)
(
  (Goto 3 5)
)
)
(BLOCK 4
(2)
(5)
(
  (Write 4 1
    (LoadLit 4 1 0)
  )
)
)
(BLOCK 5
(3 4)
(6)
()
)
(BLOCK 6  "<<< The Exit block
(5)
()
()
)
)
C The quadruple and triple code

There is a more-or-less one-to-one correspondence between the treecode and the quadruple and the triple instruction-sets. For example, they all have a BinExpr instruction:

<table>
<thead>
<tr>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BinExpr Pos Op Type Left Right) In the treecode intermediate representation, BinExpr is a node in an expression tree that computes Left Op Right. Op is a string. Type is a symbol number. Left and Right are expression trees.</td>
</tr>
<tr>
<td>(BinExpr Pos Op Type Result Left Right) In the quadruple intermediate representation, BinExpr is a three-address code instruction that computes Left Op Right. Op is a string. Type is a symbol number. Result, Left and Right are the symbol numbers of the symbols (often temporary variables) representing the instruction operands.</td>
</tr>
<tr>
<td>(BinExpr Pos Op Type Left Right) In the triple intermediate representation, BinExpr is an instruction that computes Left Op Right. Op is a string. Type is a symbol number. Left and Right are indices into the instruction stream, representing triples that compute the instruction operands.</td>
</tr>
</tbody>
</table>

The difference is in how the arguments to the binary operation are encoded. In the treecode they are pointers to sub-trees. In the quadruple code they are references to temporary variables in the symbol table. In the triple code they are integers referencing other triples.

For example, the program

PROGRAM P;
BEGIN
  WRITE 44+55;
END.

will look like this in the three different intermediate forms:

> luca_lex 9.gus | luca_parse | luca_sem | luca_AST2tree

  (11 ProcedureSy $MAIN 4 0
   ()
   ()
   0 0)
  )
  (Version 4 7 4)
  (ProcBegin 4 11)
  (Write 3 1
   (BinExpr 3 "+" 1
    (LoadLit 3 1 44)
    (LoadLit 3 1 55)
(11 ProcedureSy $MAIN 4 0
 ()
 ()
 0 0)
(12 TempSy @0 3 -1 1 1 -1)
(13 TempSy @1 3 -1 1 1 -1)
(14 ConstSy %0 3 -1 1 1 44)
(15 TempSy @2 3 -1 1 1 -1)
(16 ConstSy %1 3 -1 1 1 55)
)
(Version 4 7 4)
(ProcBegin 4 11)
(LoadLit 3 1 44)
(LoadLit 3 1 55)
(BinExpr 3 "+" 1 12 13 15)
(Write 3 1 12)
(ProcEnd 4 11)
)

> luca_lex 9.gus | luca_parse | luca_sem | luca_AST2tree | luca_tree2quad
(
(11 ProcedureSy $MAIN 4 0
 ()
 ()
 0 0)
(Version 4 7 4)
(ProcBegin 4 11)
(LoadLit 3 1 13 14)
(LoadLit 3 1 15 16)
(BinExpr 3 "+" 1 12 13 15)
(Write 3 1 12)
(ProcEnd 4 11)
)

> luca_lex 9.gus | luca_parse | luca_sem | luca_AST2tree | luca_tree2triple
(
(11 ProcedureSy $MAIN 4 0
 ()
 ()
 0 0)
(Version 4 7 4)
(ProcBegin 4 11)
(LoadLit 3 1 14)
(LoadLit 3 1 15 16)
(BinExpr 3 "+" 1 12 13 15)
(Write 3 1 12)
(ProcEnd 4 11)
)