This document specifies the LIR language and a simulator for the language. This version and the microLIR application were written by Roman Manevich.

1   LIR Language

A LIR program has the following format:

$$\{StringLiteral \mid DispatchTable\}^* \ \{Instruction\}^*$$

We now explain each of these elements.

1.1   String Literals

String literals have the format

```
name:   "text"
```

For example `str1: "In Foo_rise"`. The names of string literals can be used as addresses to integer array objects representing the string characters. These arrays are pre-allocated by the execution environment. Array names are constant global addresses; they can be accessed in any procedure but they cannot be assigned new values and the content of the strings cannot be modified (these are constant strings).

1.2   Dispatch Tables

Dispatch tables have the format

```
_name:   [labels]
```

where `_name` stores the address of the dispatch table, and `labels` is a comma-separated list of labels (labels are explained next), representing virtual functions that should be stored in the dispatch tables. As a convention, the name of the dispatch table of a class `_CLASS` is `_DV_CLASS`. The execution environment pre-allocates the table and fills the entries in the order specified with the addresses of the listed method labels. Dispatch table labels are constant global addresses that cannot be assigned new values, and the table contents cannot be modified.

1.2.1   Labels

Program labels match the pattern "`_[a-zA-Z_0-9]+:`". For example: `_this_is_a_label:`
   Labels have addresses and can be used in implementing virtual function calls. Labels are constant global addresses; they can be accessed in any procedure but they cannot assigned new values.

1.3   Instructions

LIR uses two-operand instructions. As a general constraint – an instruction may only take one operand which involves a memory access. A summary of LIR instructions is shown in Table 1 and Table 2. In these tables, we use the term "Memory" to denote a name of a program variable (local variable or parameter), "Reg" to denote a LIR register, and "Immediate" to denote a constant value. Parameters stand for anything that indicates a value — a register (Reg), a variable (Memory), a constant (Immediate), a string name (indicating the string address), or a label (indicating its address). An immediate can be any 32-bit (signed) integer.

Register names should start with a capital R. Variables are legal (non-class) identifiers in IC. Registers and Memory names can be thought of as local variables; they can be accessed only in the procedure where they are first initialized (by an update instruction or if they are formal parameters of the procedure).

You can use a single `Move` instruction to move values between any combination of memory and register operands (other than memory-to-memory).

The LIR given here is a relaxation of the LIR shown in the text-book, which allows to perform some operations using a memory-operand. This is geared towards code-generation for an Intel platform.

The low-level instructions include: unary and binary instructions, copy instruction, array access instructions, length-of instruction, field access instructions, method calls, return instructions, labels, unconditional jumps, and conditional jumps branching on various conditions depending on the value resulting from the last `Compare` operation. The convention is that binary instructions of the form "OP a,b" mean b:=b OP a. You will also use a special instruction for library function calls (`Library`).

1.3.1 Runtime Checks Summary

"Fragile" instructions should be preceded by code to conduct runtime checks and code to handle erroneous situations. Your compiler is required to emit code to implement those checks and handle them by printing an error message and exiting properly. Both of these tasks can be done in either PA4 or PA5 (up to you). We suggest you consider implementing these checks only after you have finished all other tasks and made certain that your assignment is in good shape.

Operation	Runtime Check
a[i]	`__checkNullRef(a)`
	`__checkArrayAccess(a,i)`
a.length	`__checkNullRef(a)`
new T[n]	`__checkSize(n)`
o.f	`__checkNullRef(o)`
o.m()	`__checkNullRef(o)`
a / b	`__checkZero(b)`

The following are the textual error messages associated with the runtime checks:

```
Runtime Error: Null pointer dereference!
Runtime Error: Array index out of bounds!
Runtime Error: Array allocation with negative array size!
Runtime Error: Division by zero!
```

1.4 Comments

Lines beginning with # are comments. We encourage the use of comments to document where procedures begin and end and for documenting IC statements. For example:

```
# x = y.n
Move y,R1
MoveField R1.2,R2
Move R2,x
```

1.5 Dynamic Allocation

Dynamic allocation of objects and arrays is achieved via library functions.

Table 1: LIR Instruction Summary 1

Instruction	Op1	Op2	Meaning
Data Transfer Instructions			
Move	Immediate Reg Memory	Reg Memory	Move value between registers and memory, registers and registers, and immediate to register (memory to memory is illegal)
MoveArray	Reg[Reg] Reg[Immediate]	Reg	Array access (load)
	Reg Immediate	Reg[Reg] Reg[Immediate]	Array access (store)
MoveField	Reg.Reg Reg.Immediate	Reg	Field access (load)
	Reg Immediate	Reg.Reg Reg.Immediate	Field access (store)
ArrayLength	Memory Reg	Reg	Array/String length Op1 is array/string address Op2 stores the (returned) length
Arithmetic Instructions			
Add	Immediate Memory Reg	Reg	Add registers, memory and register, or immediate and register
Sub	Immediate Memory Reg	Reg	Subtract registers, memory and register, or immediate and register
Mul	Immediate Memory Reg	Reg	Multiply registers, memory and register, or immediate and register
Div	Immediate Memory Reg	Reg	Divide registers, memory and register, or immediate and register
Mod	Immediate Memory Reg	Reg	Remainder (modulus) registers, memory and register, or immediate and register
Inc	Reg		Increment register (by one)
Dec	Reg		Decrement register (by one)
Neg	Reg		Negate register
Logical Instructions			
Not	Reg		Bitwise negation of a register
And	Immediate Memory Reg	Reg	Bitwise-and of registers, memory and register, or immediate and register
Or	Immediate Memory Reg	Reg	Bitwise of registers, memory and register, or immediate and register
Xor	Immediate Memory Reg	Reg	Bitwise-xor of registers, memory and register, or immediate and register
Compare	Immediate Memory Reg	Reg	Compare values Compare = Op2-Op1

Table 2: LIR Instruction Summary 2

Instruction	Op1	Op2	Meaning
Control Transfer			
Jump	label		Unconditional jump
JumpTrue	label		Jump when true
JumpFalse	label		Jump when false
JumpG	label		Jump greater
JumpGE	label		Jump greater-or-equal
JumpL	label		Jump less than
JumpLE	label		Jump less-than-or-equal
Library	func-name(params)	Reg	Call library function and get return value in Reg
StaticCall	func-name({Memory=param}*)	Reg	Call static function and get return value in Reg
VirtualCall	Reg.Immediate({Memory=param}*)	Reg	Call virtual function and get return value in Reg
Return	param		Use the parameter as the returned value and return from call

Operation	Library Function	
new T()	__allocateObject(s)	s is an immediate representing the object size in bytes (fields + DVPTR)
new T[n]	__allocateArray(s)	s is an operand (immediate/register/memory) representing the number of array elements

The size of an object should be the number of fields plus 1 for the dispatch vector pointer. The value passed to __allocateObject should be multiplied by 4, since the function expects to receive the number of bytes required for the object.

Another important issue is updating the DVPTR field when a new object is allocated so that it is possible to perform calls to its methods. Here is an example of LIR instructions for allocating an object of class A with 2 fields:

```
# x = new A()
Library __allocateObject(12),R1
MoveField _DV_A,R1.0
Move R1,x
```

1.6   Procedures

Procedures are simply lists of LIR instructions preceded by a label. The label can be used by static calls and as an entry in a dispatch table for virtual calls. Execution starts from the procedure labeled _ic_main. There must be exactly one such label in a LIR program.

1.7   Procedure Calls and Returns

There are three types of procedure calls: virtual calls, static calls, and library calls:

Library Function Calls.   Library functions include the functions in the libic.sig and the stringCat function. The library call instruction receives a comma-separated list of parameters that indicate the values of the arguments to be passed to the function and a register to receive the value optionally returned from the function. For example, Library __println(str2),R2 receives the name of a string literal (its address) and prints the string literal at that address.

Static Function Calls. Calls to static functions consist of the function name, a comma-separated list of pairs, and a return value register. The list of pairs are of the form "Memory=param" where "Memory" stands for the name of the formal parameter and "param" is the actual parameter. For example, let `foo` be a static function in class `C`, declared as `static int foo(int x, boolean y)`. Then the call `w=C.foo(5,z)` can be translated into the LIR instructions

```
StaticCall _C_foo(x=5,y=z),R1
Move R1,w
```

Virtual Function Calls. Virtual function calls do not name the method directly. Instead the first register (left to the dot) indicates the address of the receiver object, and the second register (to the right of the dot) indicates the offset of the method in the object's dispatch table. The offset is given in terms of the number of integers — not the number of bytes. A detailed example of virtual function calls is found in the file `test_virtual_calls` under the `test` sub-directory of the microLIR application.

Function Returns. The LIR language specified here contains a `Return` function that uses a specified parameter as the actual returned value. A simplifying assumption we make is that every function can possibly return a value. In case of functions with a `void` return type, we suggest the following convention. The last instruction of function should be `Return 9999`. The exact value is not important but can be used to quickly identify (erroneous) situations where this value is used. When a function with `void` return type is called, a designated `Rdummy` register could be used as the last operand of the call instruction. The translation from LIR to assembly (in PA5) can safely ignore this register as an optimization.

1.7.1   Additional Library Functions

You will use an additional library function (will be supplied to you) to implement '+' between strings:

Function	Meaning
`__stringCat(s,t)`	concatenate strings s and t

2   microLIR Simulator

microLIR is a simulator for the LIR langauge. It is supposed to test your translation from IC to LIR before attempting to translate from LIR to the Intel assembly language. The simulator is a Java program that accepts a file containing a LIR program and executes the program.

2.1   Downloading and Installing the Simulator

The simulator can be downloaded from the course web site. Installation consists of simply unzipping the archive into your chosen installation directory. The installation contains Java sources, .class files, and several LIR sample programs (under the `test` sub-directory).

2.2   Program Usage

The program usage is:

```
Usage: file [options]
options:
  -printprog            Prints the program
  -verbose:num          Execution verbosity level (default=0)
                        0 - quiet mode
                        1 - announce next instruction to be executed
                        2 - announce instructions and computed values
  -stats                Prints program statistics and exits
```

In order to invoke the program enter `java -cp <LIR_INSTALLATION_PATH>/build microLIR.Main <args>` or add the `build` sub-directory to the CLASSPATH and enter `java microLIR.Main <args>`. The simulator supports all LIR instructions and most library functions. A batch file `microlir.bat` enables activating the program from a Windows environment.

2.3 Limitations

The simulator has undergone only very basic testing. If you believe you have found a bug, please send us the input causing the problem with a clear and detailed explanation. Here are some known limitations:

- The simulator handles only simple literal strings not containing escape characters.

- The simulator does not implement the following library functions: `readln`, `eof`, `stoa`, `atos`, and `random`.

2.4 Tips and Possible Pitfalls

Debugging An erroneous LIR program may cause behavior which may seem strange (just as your final translation may do when executed on an Intel machine). The simulator is equipped with functionality to detect various problems (access to illegal addresses, division by zero, allocating objects with sizes that are not multiples of 4, etc.). However, not all problems are detected. In order to understand what is wrong in your LIR program, you may raise the verbosity level of the execution and follow the list of instructions executed and the values computed by the instructions..

main fall-through The simulator processes the instructions one after another, unaware of procedure boundaries. Therefore, if the `_ic_main` procedure is not the last procedure, the simulator may follow by executing other code, not as you intended. There are two possible solutions for this: (i) print the translation of main as the last part of the program, or (ii) add a `Library __exit(0),Rdummy` instruction as the last instruction in the main procedure.

Missing Return instructions Every procedure, **except `_ic_main`**, should end with a `Return` instruction. A missing `Return` instruction results in following the execution with the first instruction of the procedure below.

Addresses The simulator allocates addresses for labels using their position in the sequence of instructions. Addresses of allocated objects (including strings and dispatch tables) start from $30,000$. The simulator never releases memory, and thus the address of an allocated object is never returned in subsequent allocations.

Compare register The Intel architecture includes an `EFLAGS` register, which behaves similarly to the `Compare` register of microLIR. However, while the `Compare` register only updates on `Compare` instructions, the `EFLAGS` register updates on every arithmetic instruction. Therefore, your LIR program should not depend on the value of the `Compare` register after an arithmetic instruction is executed. Instead, you should aim to use the value of the `Compare` register (by one of the `JumpXX` instructions) immediately after a `Compare` instruction.

Variable Shaddowing. IC allows local variables to be hidden by defining variables with the same name in inner blocks. Therefore, each variable should be given a unique name to avoid name clashes. A possible naming scheme is the following. Local variables names can be `vIDname` where ID is a unique id number and **name** is the actual name. Similarly, parameter names can follow the pattern `pIDname` and field names can follow the pattern `fIDname` etc.