Assignment 5 Report - Optimization of compiler

By Alexander Aakersoe

Deadline - 12/07/2024 23:00

Contents

1 The Unoptimized compiler

Before any optimizations are implemented we should test the unoptimized compiler. This will both give us benchmarks to compare the improved versions of the compiler against as well as give us an idea of where to start.

1.1 Testing

All testing in the report will be done using example29 and example31 as suggested. These programs are ideal to use for testing as they are both fairly computationally dense - we will in fact be able to see if our optimizations make a difference - and also a good example of programs doing a common computation. All tests will be the average of running three tests. The tests for the unoptimized compiler can be seen below. I will only be referencing the user time as this is the benchmark we will be using.

TEST#	EXAMPLE 29	EXAMPLE 31
1	0.673s	0.692s
2	0.642s	0.700s
2	0.702s	0.730s
AVERAGE	0.672s	0.707s

Table 1: User run-times for example29 and example31 on the unoptimized compiler

1.2 Considerations for optimization

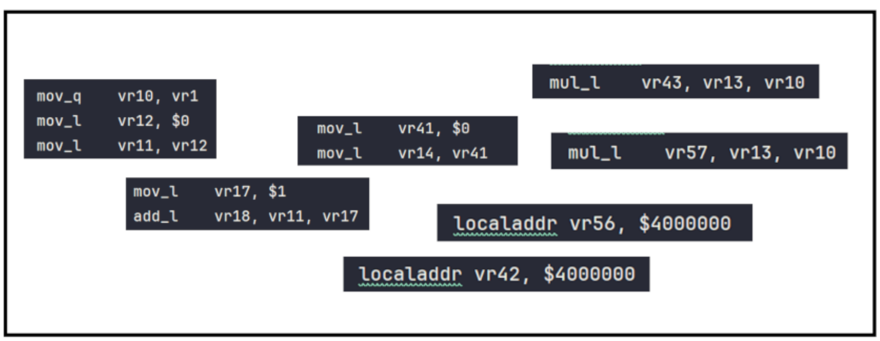

Figure 1: Assembly-snippets from example29. We see plenty of suboptimal computations - for example is the address being loaded twice, multiplication is computed when the result is already available, and registers are used as intermediate values when values could just be propagated directly.

To get an idea of what optimizations should be implemented we look at the generated assembly code. Looking at example29.S it is clear that we are doing a lot of redundant computations. These are especially linked to getting the addresses of the array (where we are adding and subtracting). To get rid of redundant computations I've elected to implement local value numbering, copy-propagation and dead-store elimination.

2 Optimization 1: Local value numbering, Copy-Propagation and dead-store elimination.

To implement Local Value Numbering we need to do bookkeeping using 5 maps and do the following:

1. Build a CFG (control flow graph) consisting of basic-blocks.
2. Within each basic block do the following:
3. Assign a number to each value
4. If the virtual registers contain the same value number then they must contain the same value
5. If a computation is repeated (same operation on same values) it is redundant and can be replaced by reuse of the result from the previous computation. In other words, replace the computation with a move instruction that copies the original result into the destination of the new computation.

To implement Copy-propagation we need to keep track of two maps and do the following:

1. Start from first instruction in BB, then for each instruction:
2. A) Check if any of the source registers are a key in VREG map - if so replace it with the associated value
3. B) Check if the destination register is part of any pair in any map - if so remove the pair from the map.
4. Whenever we encounter a MOV instruction (copying):
5. A) If both operands are registers: Add the register pair to the VREG map
6. B) If source operand is Immediate: Add the register-imm pair to the CONST map

To implement Dead-store elimination we need to make use of a liveness-analysis. This analysis gives vital information as to whether or not a register is dead immediately after assigning to it. If this is the case then we can delete the assignment instruction from the instruction sequence.

2.1 Results: Spatial changes

When implementing these optimizations my first goal was to get started on Local value numbering. Along the way I ran into many bugs, but these were resolved rather quickly. Copy propagation was then implemented and lastly Dead-store elimination was added. Running just the LVN does not make much of a difference. Neither does running the copy-propagation. It is when all three of these optimizations are run consecutively that we see real changes to the code. Below is an example of this, running my optimized compiler on example29. We are here looking at the part of the main function in which the addresses of the array-indexes are computed.

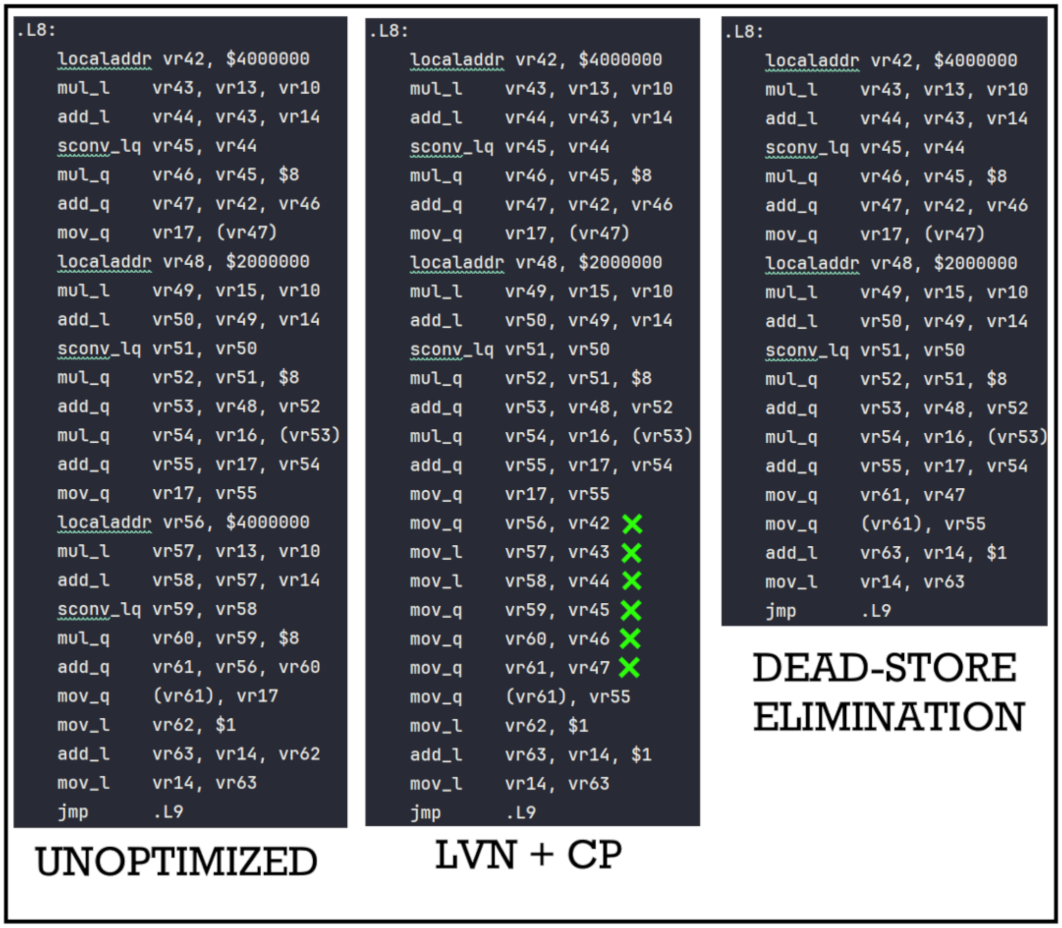

Figure 2: Testing my three optimizations on example29. We see clear changes after the Dead-store elimination has ran. It should however be noticed that we see clear changes already after LVN, however no instructions are here removed, we only exchange these with MOV instructions (copying). In total seven instructions are removed from this snippet of assembly.

2.2 Results: Temporal changes

To find out exactly how much these optimizations have done for the user-run time of the program we run the timing on both example29 and 31 for all three iterations of optimization.

2.3 Decisions made along the way

Along the way, several iterations of both LVN and copy propagation were tried and improved. As a result, a few special cases have been found and eliminated while the optimization-techniques have been expanded in general. One special case, which has now been fixed, was that copy-propagation would produce instructions as follows:

$$MOV vr0, vr0$$

These are of course nonsensical, however it is NOT the copy-propagation's job to eliminate these. This job falls upon the Dead-Store elimination algorithm. What then

LVN	EXAMPLE 29	EXAMPLE 31
1	0.663s	0.681s
2	0.639s	0.715s
2	0.658s	0.690s
AVERAGE	0.653s	0.695s
PCT IMPROVED	2.8%	1.7%
COPY-PROP		
1	0.691s	0.687s
2	0.617s	0.681s
2	0.639s	0.678s
AVERAGE	0.649s	0.682s
PCT IMPROVED	3.4%	3.5%
DEAD-STORE		
1	0.598s	0.599s
2	0.601s	0.619s
2	0.620s	0.605s
AVERAGE	0.606s	0.608s
PCT IMPROVED	9.8%	14.8%

Table 2: User run-times for example29 and example31 on the compiler with selected optimizations. Please notice that these optimizations are cumulative (COPY-PROP times are with both LVN and COPY-PROP etc.). It should be noted that the improvements for LVN and COPY-PROP is not a very good figure as the range of the user-times itself outweigh the percentage of improvement.

happens (as was the case) when the destination register is not dead? The instruction is not eliminated. Some special case handling was therefore needed, recognizing whenever an operand was moved into itself. This is an optimization which would be obvious to handle using peephole optimization - however since I've not gotten to that part I instead decided to handle it here.

An example of expanding the optimization in general was the introduction of constant propagation. This requires another map, mapping registers onto integer values. With such a map it is possible to remove registers from instructions and replace these with constants instead (best case). An example of this optimization can be seen below.

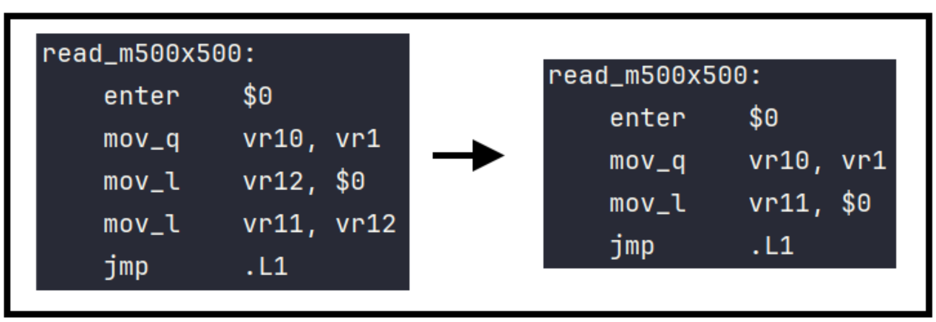

Figure 3: Example of constant-propagation from example29

2.4 Further optimizing the compiler

Much work can still be done to improve the performance of this compiler. Noticeably, we are currently storing every virtual register in the stack frame. This simplification was useful when we first created the compiler; however, it is detrimental to the running time of the assembled code as reading/writing to memory is far more time consuming than using the processor registers. I will therefore next implement Local Register Allocation. This optimization will work on the High-level IR and reshape it such that the Low-level IR can process an instruction-sequence of High-level instructions annotated with machine-registers.

3 Optimization 2: Local register allocation

3.1 Algorithm

I've decided to implement Local Register Allocation with the following specifications:

1. The following registers are always reserved: %R10, %R11, %RSP, %RBP, %RAX. R10 and R11 are used for handling high-level 3-register instructions. RSP and RBP are the stack pointer and stack frame. These are used for accessing memory and are therefore reserved. RSP could be freed if it was pushed at the beginning of every function and popped back at the end. RAX is used the return register and as such needs to be reserved for this case.

2. The following registers are reserved depending on if there is a call instruction in the Basic-block. These are written in the order of reservation and the number of reservations corresponds to the number of argument registers needed for the function call: %RDI,%RSI,%RDX,%RCX,%R8,%R9.

3. Virtual registers which are either A) alive at the beginning of the basic block, B) alive at the end of the basic block, or C) in the range from 0-10 are never assigned a virtual register. I've hence decided to store these in the stack frame. An alternative would have been to assign some of these a personal MREG and push/pop this as needed. Dataflow analysis (liveness analysis) has been used to determine what registers should be put in this list.

4. The Machine Register Queue is constructed with %R15 being first out, then %R14 and so forth. The machine registers that are reserved are simply removed from this queue before optimization begins. When a machine-register is freed it is put in the front of the queue (much like a stack).

5. If a source/destination register is dead after the current instruction (liveness analysis), and it has an assigned Machine-register, this machine register is freed. Source operands are checked before the destination operand and as such the destination operand has the ability to use this freed machine-register immediately. In fact this happens fairly often as the freed-machine register is put in front of the machine-register queue.

To construct Local-register allocation the following structures are used:

1. M_Queue - the queue used for machine registers. std::deque used.

2. reservedList - the list of reserved registers. std::vector used.

3. AssignMap - A map of Vregs(key) to Machineregs (value). std::map used.

4. spilledMap - A map of registers(key) that have been spilled, with the offset(value). std::map used.

5. spilledStorage - integer value counting the total size of spilled registers for the current basic-block. If this value is greater than the former spilledStorage value, then m_spilledStorage is updated. This provides a way of informing the LL-IR of how much memory storage it should expect to allocate for spills.

The Algorithm is as follows:

1. build M_Queue. Build reservedList.

2. For each instruction:

3. For each register operand (VREG or VREG-based MEMREF), if the register is not in the reservedList do the following:

4. If the register is spilled, then allocate a machine-register to restore it to and restore it. Update spilledMap (remove the register). Update AssignMap (add the pair). Update the Queue (pop the front Machine-Register).

5. Else, if the register is not a key in AssignMap, then allocate a machine-register*. If the register is in AssignMap instead copy the Machine-register value from AssignMap. update data structures as needed.

6. Annotate the operand with the machine-register.

7. If the register is dead after the instruction (liveness analysis), then erase the key-value pair from AssignMap.

8. Emit Annotated high-level instruction to instruction-sequence.

9. * If the Machine Register Queue is empty, then we must spill another register. The algorithm for doing this is below.

In the case that a register must be spilled, we use the following algorithm.

1. Traverse the annotated instruction-sequence from the top.

2. For each instruction, examine the operands. When we encounter an operand with a register currently in AssignMap, Steal the machineRegister. Put the pair of (victimRegister, offset) in spilledMap. Update AssignMap. Update spilledStorage. Break, Return the machineRegister.

Selecting the register which is furthest towards the top of the basic-block must also be the register which is used least frequently.

3.2 Testing

3.3 Spatial Changes

The Local-register Allocation algorithm is not destructive. Hence no instructions in the High-level IR are changed meaningfully (besides from spills and restores which are added). We do however see clear changes to the LL-IR. We continue the example from example29 to compare. Notice the comments are the HL-IR annotated with LL-MachineRegisters.

```
.L8:
 leaq     -2000000(%rbp), %r10 /* localaddr vr42<%r15>, $4000000 */
 movq     %r10, %r15
 movl     -6000040(%rbp), %r10d /* mul_l    vr43<%r14d>, vr13, vr10 */
 imull    -6000064(%rbp), %r10d
 movl     %r10d, %r14d
 movl     %r14d, %r10d        /* add_l    vr44<%r14d>, vr43<%r14d>, vr14 */
 addl     -6000032(%rbp), %r10d
 movl     %r10d, %r14d
 movl     %r14d, %r10d        /* sconv_lq vr45<%r14d>, vr44<%r14d> */
 movslq   %r10d, %r10
 movq     %r10, %r14
 movq     %r14, %r10          /* mul_q    vr46<%r14>, vr45<%r14>, $8 */
 imulq    $8, %r10
 movq     %r10, %r14
 movq     %r15, %r10          /* add_q    vr47<%r15>, vr42<%r15>, vr46<%r14> */
 addq     %r14, %r10
 movq     %r10, %r15
 movq     (%r15), %r14        /* mov_q    vr17<%r14>, (vr47)<%r15> */
 leaq     -4000000(%rbp), %r10 /* localaddr vr48<%r13>, $2000000 */
 movq     %r10, %r13
 movl     -6000024(%rbp), %r10d /* mul_l    vr49<%r12d>, vr15, vr10 */
 imull    -6000064(%rbp), %r10d
 movl     %r10d, %r12d
 movl     %r12d, %r10d        /* add_l    vr50<%r12d>, vr49<%r12d>, vr14 */
 addl     -6000032(%rbp), %r10d
 movl     %r10d, %r12d
 movl     %r12d, %r10d        /* sconv_lq vr51<%r12d>, vr50<%r12d> */
 movslq   %r10d, %r10
 movq     %r10, %r12
 movq     %r12, %r10          /* mul_q    vr52<%r12>, vr51<%r12>, $8 */
 imulq    $8, %r10
 movq     %r10, %r12
 movq     %r13, %r10          /* add_q    vr53<%r13>, vr48<%r13>, vr52<%r12> */
 addq     %r12, %r10
 movq     %r10, %r13
 movq     -6000016(%rbp), %r10 /* mul_q    vr54<%r13>, vr16, (vr53)<%r13> */
 imulq    (%r13), %r10
 movq     %r10, %r13
 movq     %r14, %r10          /* add_q    vr55<%r14>, vr17<%r14>, vr54<%r13> */
 addq     %r13, %r10
 movq     %r10, %r14
 movq     %r15, %r15          /* mov_q    vr61<%r15>, vr47<%r15> */
 movq     %r14, (%r15)        /* mov_q    (vr61)<%r15>, vr55<%r14> */
 movl     -6000032(%rbp), %r10d /* add_l    vr63<%r15d>, vr14, $1 */
 addl     $1, %r10d
 movl     %r10d, %r15d
 movl     %r15d, -6000032(%rbp) /* mov_l    vr14, vr63<%r15d> */
 jmp      .L9                 /* jmp      .L9 */
```

Looking at the LL-IR and the annotated HL-IR, we see a lot of the characteristics which was discussed earlier. For example, many of the HL instructions use the same machine-register as a source and destination operand. The instructions from figure 1 are completely conserved and annotized. Many of the HL-instructions result in more than one LL-instruction. Even though many of these would be easy to optimize away, the LRA does not have this capability. For this we would need an algorithm which could do optimizations on the LL-IR. One such technique could be peephole optimization. Below is another example.

```
.L10:
 leaq     -2000000(%rbp), %r10 /* localaddr vr63<%r15>, $4000000 */
 movq     %r10, %r15
 movl     -6000040(%rbp), %r10d /* sconv_lq vr64<%r14d>, vr11 */
 movslq   %r10d, %r10
 movq     %r10, %r14
 movq     %r14, %r10          /* mul_q    vr65<%r14>, vr64<%r14>, $4000 */
 imulq    $4000, %r10
 movq     %r10, %r14
 movq     %r15, %r10          /* add_q    vr66<%r15>, vr63<%r15>, vr65<%r14> */
 addq     %r14, %r10
 movq     %r10, %r15
 movl     -6000032(%rbp), %r10d /* sconv_lq vr67<%r14d>, vr12 */
 movslq   %r10d, %r10
 movq     %r10, %r14
 movq     %r14, %r10          /* mul_q    vr68<%r14>, vr67<%r14>, $8 */
 imulq    $8, %r10
 movq     %r10, %r14
 movq     %r15, %r10          /* add_q    vr69<%r15>, vr66<%r15>, vr68<%r14> */
 addq     %r14, %r10
 movq     %r10, %r15
 movq     (%r15), %r13        /* mov_q    vr15<%r13>, (vr69)<%r15> */
 leaq     -4000000(%rbp), %r10 /* localaddr vr70<%r12>, $2000000 */
 movq     %r10, %r12
 movl     -6000024(%rbp), %r10d /* sconv_lq vr71<%r9d>, vr13 */
 movslq   %r10d, %r10
 movq     %r10, %r9
 movq     %r9, %r10           /* mul_q    vr72<%r9>, vr71<%r9>, $4000 */
 imulq    $4000, %r10
 movq     %r10, %r9
 movq     %r12, %r10          /* add_q    vr73<%r12>, vr70<%r12>, vr72<%r9> */
 addq     %r9, %r10
 movq     %r10, %r12
 movq     %r12, %r10          /* add_q    vr76<%r12>, vr73<%r12>, vr68<%r14> */
 addq     %r14, %r10
 movq     %r10, %r12
 movq     -6000016(%rbp), %r10 /* mul_q    vr77<%r12>, vr14, (vr76)<%r12> */
 imulq    (%r12), %r10
 movq     %r10, %r12
 movq     %r13, %r10          /* add_q    vr78<%r13>, vr15<%r13>, vr77<%r12> */
 addq     %r12, %r10
 movq     %r10, %r13
 movq     %r15, %r15          /* mov_q    vr85<%r15>, vr69<%r15> */
 movq     %r13, (%r15)        /* mov_q    (vr85)<%r15>, vr78<%r13> */
 movl     -6000032(%rbp), %r10d /* add_l    vr87<%r15d>, vr12, $1 */
 addl     $1, %r10d
 movl     %r10d, %r15d
 movl     %r15d, -6000032(%rbp) /* mov_l    vr12, vr87<%r15d> */
 jmp      .L11                /* jmp      .L11 */
```

I've decided to include this example as it shows that my version of the LRA has less of a need of spills and restores than the reference solution. This boils down to more registers being available (as local variables do not get assigned personal registers) and copy-propagation. The registers %R15, %R14, %R13 %R12, %R9 are used dynamically. Compared to the register solution, which also uses five registers dynamically, but also reserves a total of five "personal" registers in addition to the normally reserved registers.

3.4 Temporal Changes

To find out how much LRA has done for the user-run time of the program we run the timing on example29 and example31. This will be done both without the other optimizations (to get a stand-alone benchmark) and with the other optimizations. Comparing these results to the ones given as part of the assignment shows a fairly large discrepancy. On the bright side, the user-time without any optimization benchmark is far lower (probably due to a stronger processor), however the effect of the optimization is much smaller. Using all of the optimizations so far result in a 53.1% improvement of the user-time, where as the reference solution has a much higher (82.5%) improvement rate. It is however still within an acceptable range. The main differences are probably due to the reference solution's way of dealing with Local-register-allocation. As we saw, I did not need to odo any spills or restores, however my code is not as optimal. Using "personal" registers for local variables which are often invoked mean making far fewer accesses into the stack-frame, which saves a lot of time.

NO OPTIMIZATIONS	EXAMPLE 29	EXAMPLE 31
1	0.673s	0.692s
2	0.642s	0.700s
2	0.702s	0.730s
AVERAGE	0.653s	0.695s
ONLY LRA		
1	0.427s	0.484s
2	0.421s	0.477s
2	0.421s	0.474s
AVERAGE	0.423s	0.478s
PCT IMPROVED	37.5%	32.3%
FULL OPTIMIZED		
1	0.315s	0.333s
2	0.325s	0.323s
2	0.308s	0.341s
AVERAGE	0.316s	0.332s
PCT IMPROVED	53.1%	53.0%

Table 3: User run-times for example29 and example31 on the compiler with Local-register Allocation, and with all optimizations.

3.5 A lesson learned along the way - debugging

A lot of time was spent on this assignment on debugging my code. One issue was especially frustrating - example 29 and 31 would not run properly. I checked my optimizations algorithm repeatedly and could not find any issues. Running the program without LRA optimization was possible, so I concluded the issue must lie here. This turned out NOT to be the case. The program broke when LRA was added, because my way of handling the add-instructions in the LL-program were not conservative. In other words, the add-instruction would impact not only the destination operand, but also one of the source operands (one source operand would be added to the other and then the result moved into the destination). The copy-propagation algorithm had no knowledge of this (why would it, it operates on the HL-IR!) and as such values were propagated through the program which were in fact not equal. Spending this much time on debugging stopped me from implementing future optimizations, which would have been cool!

As an electro-engineering student, I've not spent much time writing code (especially not C++). A lesson I will for sure take to heart in my future projects is to make sure that you start from a place of correctness and test profusely along the way :)

3.6 Conclusion

To conclude on this small project (assignment?), Local-Value Numbering, Copy-propagation, Dead-Store Elimination, and Local-register Allocation has been implemented. Each of these optimizations have had an impact on both the spatial domain (minimizing the number of instructions) and the temporal domain. Most noticeably, Local-register allocation has proved the most effective at reducing the user-time. Benchmarked against the time to run example29 and example31 without optimizations, LRA showed a decrease in user-time by 37.5%, with the benchmark being 53.1% using all optimizations. Dead-store elimination proved most useful of the optimizations when it comes to reducing the amount of instructions. Future improvements should be to add peephole-optimization. This can be done on both the HL-IR and LL-IR. From the examples used in this report it seems that targeting the LL-IR could be extremely effective.

3.7 Final notes

It should finally be noted, that:

1. My optimized version of the compiler does not work for example33. I sadly have not had the time to debug and correct this issue, however it does not seem to be an issue with my optimizations as the code fails both with and without these.

2. not being a software-engineering student has proved to be a small issue. Currently running the program will only work for the optimized version. To run the code without being optimized, uncomment line 139 of "lowlevel_codegen.cpp". (This is necessary as the stack-frame being used in the LL-IR is different between optimized and unoptimized versions.