# 1. Completion Status

We implemented expression tree building, grammar construction, register allocation, and code generation. We have tested our implementation on some our self-designed test cases and it works functionally. Unfortunately, due to the tight schedule, we are not able to test them thoroughly.

## 1.1 Assumptions

- All values are 64 bits long integers.
- No undefined values in the program.
- %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rbp, %rsp, %r8, %r9 are treated as special registers
- Stack capacity is unlimited (no need to pop values pushed to the stack).

## 1.2 Features

The most codes generated for these features pass eyeball test.

- Support local 64-bit integer variables and arguments.
- Integer arithmetic operations include add, sub, multiply.
- Single dimensional array. We have arrays implemented, but we do not handle arrays with operands of constant expression type.
- Conditional and unconditional branch operations.
- Loops. Loops are supported, but we have not implemented break, continue, and PhiNode yet.
- Function calls. Our function call is able to support non-fixed number of arguments and recursion. For standard library function, we have the problem that llvm IR has replaced standard library with llvm function intrinsics (funtion starts with prefix "llvm"). If this can be bypassed, library functions should be supported.

Register allocation works functionally as an separate component since we have not fully implemented its integration to previous components. For the running result of register allocation in each test case, please check out the standard out stream.

# 2. Building Expression Tree

## 2.1 Expression Tree Structure

Our tree structure is designed mostly after the tree structure in `sample4.brg`. We consider the root of an expression tree (for data flow) to be preserved either in register, or in memory, whereas the intermediate tree nodes are temporaries. Internally, whether a value will be preserved or not depends on the reference counter, which will be introduced in the later text.

We also made some other changes to the original tree structure. we used a vector to store the kids so we can potentially have unlimited number of children in a tree. This design fits most instructions well since it provides the ability for supporting different number of operands. Another important addition to the original tree structure is that we added the `refcnt` – reference counter, to each tree node, so we know how many times a tree node has been used as an operand during code generation.

The details of expression tree structure are in

- `Tree.h`

### 2.2 Expression Tree Building

In expression tree building, we adopted an bottom-up approach. We iterate through each instruction in each basic block, and build an expression tree for each instruction, then store it in a map. For each instruction, we find the definition of each operand, and use the definition as a key to find the sub-expression trees, then we can attach the sub-expression trees as kids to the current expression tree that we are building. At the same time, the sub-expression tree being referenced will increment their reference counter by 1. On finish building the expression tree, we add it into a vector/list for code generation.

The above procedure describe the general cases of expression tree building. There are some circumstances that an operand cannot be dynamically casted to an instruction. For example, an operand cannot find its definition in previous instruction the following two cases: 1. `Br` instruction, the operands contain labels to basic blocks. 2. Argument, any instruction could possibly use an argument as an operand. 3. Constant, an immediate number cannot find its definition because there is no definition. In order to solve for circumstances 1 and 2, we first adopt an iteration to record the basic blocks and arguments, and then store them into separate maps for indexing. To solve the circumstance 3, we simply create a tree with type `IMM` and set the value to be the constant.

The details of expression tree building are in

- `llc_olive_helper.cxx:BasicBlockToExprTrees(...)`
- `llc_olive_helper.cxx:MakeAssembly(...)`

## 3. Grammar Construction

### 3.1 Terminals

Our grammar defines all llvm instruction types as terminals, such as `Ret`, `Br` and `Call`, etc. In our design, the opcode for these terminals are exactly the same as the opcode we get in llvm::Instruction. Such alignment makes it easy for grammar matching. In addition, we add `DUMMY`, `REG`, `IMM`, `MEM`, `LABEL`, `ARGS`, and `NOARGS`. These additional labels will be introduced in grammar rules.

### 3.2 Non-terminals & Grammar

- `stmt`: This is the starting symbol of the grammar.

- `reg`: This rule matches instructions that will output to a register, such as adding to register, subtracting from register, etc. Arithmetic instructions and load operations are defined as `reg`.
- `imm`: This rule matches immediate numbers, and instructions that is equivalent to generating an immediate number, such as loading an immediate number to register.
- `mem`: This rule matches instructions that will output to a memory address, such as adding a register to memory address, or subtracting a register from memory address, etc. Store and Alloca operations are defined as `mem`.
- `cond`: This rule matches only cmp instruction as branch condition. In llvm, the only possible instructions are ICmp and FCmp, for integers and floating point respectively.
- `label`: This rule matches only the name of basic blocks that are used in a branch instruction.
- `args`: This rule matches the arguments of functions. It could either NOARGS, or ARGS, representing no more argument, or more arguments to come. Since grammar rule requires each non-terminal rule to have a fixed arity, `args` is constructed as a binary tree. In order to transform vector into binary tree, we call Tree::KidsAsArguments() to perform such transformation.
- `value`: This rule matches all possible values: registers, immediate numbers and memory addresses.
- `ri`: This rule matches registers and immediate numbers.
- `rm`: This rule matches registers and memory addresses.

The flexibility in llvm IR causes some troubles in grammar definition:

- llvm IR allows the same instruction to have different arity. For example, `Br` instruction could either take one operand or three operands, representing unconditional jump and conditional jump, respectively, but our grammar definition in olive requires a rule associated with the same terminal must have the same arity. As mentioned before, our terminals are aligned exactly with llvm IR instruction opcodes, therefore it is not appropriate for us to create another terminal. The way we solve this problem is by padding the arguments with DUMMY. In the `Br` example, we will pad the unconditional jump with two DUMMY nodes, so the arity of two kinds of jumps aligned again.
- llvm IR allows var_arg for Call instruction. For a function call, the number of operands in a `Call` instruction is undetermined. Therefore it is impossible to write a plain rule to match the operands verbatimly. Our way to solve this problem is by create recursive grammar rules for matching arguments. Arguments are first transformed into a binary tree before passed on to olive, then olive could match each `args` with either `ARGS(value, args)` recursively, or `NOARGS`, the terminator of arguments.

The details of grammar definition are in

- `llc_olive.brg`

### 3.3 Costs

The cost of each operation is defined based on the sum of the cost of operands, plus some constant numbers we choose for the instruction. For memory and branch operations, we give a higher cost. For register operations, we give a relatively lower cost.

## 4. Register Allocation

The Poletto's paper [1] describes a general framework of allocating limited number of physical registers to arbitrary number of live ranges obtained from liveness analysis. Later research found that the register allocation could largely benefit from Static Single Assignment (SSA) form. In this project, we implemented the SSA-form-based register allocation originated in the paper [2].

For the following part, we will present the details of our implementation in three steps: Building Lifetime Intervals, Allocating Registers, and Integrating Allocation to Code Generation.

### 4.1 Building Lifetime Intervals

The very first step is to build lifetime intervals in terms of the SSA-form LLVM intermediate representation. The objective of doing so is to generate one lifetime interval for each virtual register, covering operation numbers where the register is alive and with lifetime holes in between. Such lifetime interval contains the usage information of virtual registers and thus serves as the input of physical register allocation in later step.

We initiated two classes `class Interval` and `class LiveRange` to represent the corresponding interval and included live ranges of a virtual register respectively. In `class Interval`, we correctly implemented two operations required by the SSA-form-based interval construction:

- `Interval::addRange(int from, int to)`: impose the live
- `Interval::setFrom(int pos)`: shorten one live range of the interval and change the start-point of the live range that `pos` resides in to `pos`

Note that `addRange` needs to merge live ranges if there exist overlapping ones and we correctly implemented (see log in stdout).

The details of these two significant data structures can be found in

- `LiveRange.h`:

Here is the procedural description of our way of constructing lifetime intervals (see Figure. 1). Given the intermediate representation in SSA form, we traverse each block `b` in reverse order where all dominators of a block are before this block, and where all blocks belonging to the same loop are contiguous. For each block `b`, we initialize the `live` as the union of all its successors' live virtual registers. Then we add input operands of all phi functions residing in `b`'s successors to `livein`. And now all intervals in existing `live` will be assigned a new range of current block. Afterwards, we iterate through each individual instruction within the current block `b`, updating all intervals corresponding to each input and output operand of the instruction, and removing/adding virtual registers from/to the `live`. Next, we need to remove all corresponding virtual registers of output operands of all phi function in current block `b`. Lastly, if the current block `b` is a loop header,

```
BuildIntervals
for each block b in reverse order do
    live = union of successor.liveIn for each successor of b

    for each phi function phi of successors of b do
        live.add(phi.inputOf(b))

    for each opd in live do
        intervals[opd].addRange(b.from, b.to)

    for each operation op of b in reverse order do
        for each output operand opd of op do
            intervals[opd].setFrom(op.id)
            live.remove(opd)
        for each input operand opd of op do
            intervals[opd].addRange(b.from, op.id)
            live.add(opd)

    for each phi function phi of b do
        live.remove(phi.output)

    if b is loop header then
        loopEnd = last block of the loop starting at b
        for each opd in live do
            intervals[opd].addRange(b.from, loopEnd.to)

    b.liveIn = live
```

Figure 1: Build Lifetime Intervals Based on SSA-form Intermediate Representation

```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) = R then
            SpillAtInterval(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldIntervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

Figure 2: Overall Framework of The Linear-Scan Register Allocator

all intervals in existing `live` should be appended with a new range from the start of the block to the end of the loop. Before steping into the next basic block, move all remained virtual registers in `live` to `livein`.

The details of building lifetime intervals can be found in

- `llc_olive_helper.cxx:BuildIntervals(...)`

## 4.2 Allocating Physical Registers

Our implementation strictly follow the general framework of physical register allocation described in the Poletto's paper [1] as shown in Figure 2.

We created one class named `class RegisterAllocator` to encapsulate all elementary data structures and operations required by the task of register allocation. The basic idea of our linear-scan register allocator is that it allocates one free register when one available register exists, and spill one interval (virtual register) out to stack if all physical registers are occupied.

Two fundamental procedures in determining which specific physical registers for allocation:

- `tryAllocateFreeReg()`: allocate free register to the new-coming interval when there is one available register.
- `allocateBlockedReg()`: spill one interval and allocate its register to new-coming interval.

For these two procedure, we strictly follow the strategy described in the paper [2] as shown in Figure. 3.

One noticeable variation in our implementation is that when deciding `freeUntilPos` and `nextUsePos`,

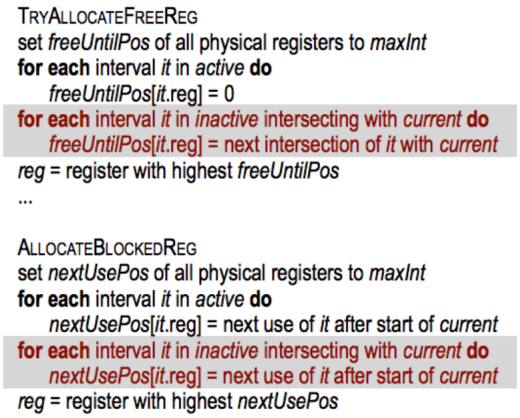


Figure 3: Strategy of Register Allocation

we take the minimal of all conflicting next intersection (or next use), which causes very reasonable choice of physical register.

Other details about `class RegisterAllocator` can be found in

- `RegisterAllocator.h`
- `RegisterAllocator.cpp`

### 4.3 Integrating Allocation to Code Generation

Acquiring the generated allocation result at the stage of code generation is another tricky thing of this project. The way we access the allocation result is as follows. For each *instruction or value* involved in the register replacement (from virtual to physical)

- Look up its associated virtual register.
- Look up its operation number in LLVM intermediate representation.
- Use the associated virtual register to access the value's corresponding interval.
- Scan through all live ranges of its interval and determine which live range of its interval the operation number resides in.
- Look up which physical register has been assigned to it and output that physical register.

At this point, the integration part does not work because we bifurcate our implementation and have different virtual register coding. If given more time, we would be able to resolve this problem.

## 5. Code Generation

### 5.1 Select Which Expression Tree to Build

Since we start building our expression tree using a bottom-up approach, we end up with many expression trees. Therefore we need to decide which expression trees to be thrown into olive for

code generation. In our case, we use refcnt to decide whether an expression tree should be turned into codes.

By traversing the list of expression trees, if the refcnt of an expression tree is 0, then we consider it to be the root of an expression tree, and therefore feed it into olive for code generation. If the refcnt of an expression tree is not 0, then it must be used as an operand somewhere in other trees. Therefore we ignore it.

In our observation, instructions such as Store and Branch are always the root of an expression tree, whereas instructions like Alloca will almost always be referenced in some other instruction.

The details of code generation in this part are in

- `llc_olive_helper.cxx:MakeAssembly(...)`

### 5.2 Fake Assembly Code Generation

We perform the fake assembly code generation in olive actions. Our fake assembly codes will fill every register with virtual registers, denoted in the form %0, %1, etc. All olive actions take in a `FunctionState` object, which manages the virtual register assignment, and provides convenient interfaces for fake assembly code generation. Each olive action will generate one or more `X86Inst` objects to represent an X86 instruction, and it will add the generated instructions into the `FunctionState` object in sequence. All the added instructions will be printed after register allocation.

There are several differences in llvm IR to X86 assembly translation, causing the translation to be hard. They are

- llvm IR does not have limit in arity, but X86 assembly uses at most two operands in one instruction. llvm IR is a relatively high level representation compared to X86, since one instruction in llvm IR could result in multiple X86 assembly instructions after translation.
- llvm IR outputs the result into a new virtual register, while X86 output the result into the `dst` operand. This implies that the virtual registers in llvm IR and the virtual registers in fake assembly are different. To solve this problem, we provided `CreateVirtualReg(...)` in `FunctionState` class to allocate a new virtual register, and then we can use this newly allocated virtual register in the generated X86 instruction. However, one might not always need a new virtual register if one of the operands has only 1 reference, because that means the operand will not be used anymore, so it will be safe to directly use its virtual register. To implement this condition, we also provided `AssignVirtualReg(...)` in `FunctionState` to directly assign virtual register from existing node.
- llvm IR does not use special registers, but X86 assembly does. llvm IR is a universal intermediate representation that does not involve physical registers. However, in X86 assembly, physical registers are used in multiple cases, such as return value, function call parameters, stack frame, etc. To compensate for this, we also provided `CreatePhysicalReg(...)` in `FunctionState`. Registers created using this method will bypass the register allocation process. Therefore, this method should only be used to create special-purpose registers, otherwise it will cause mistakes in the generated program.

The details of fake assembly generation are in

- `llc_olive.brg`
- `FunctionState.h`

### 5.3 Print X86 Assembly

We provided `PrintAssembly(...)` in `FunctionState` for assembly printing. In this function, we will print the prolog, the function body and the epilog of the function. We overrided the print operator in `X86Inst` so that it will print the physical register allocated by the reigster allocator.

The details of fake assembly generation are in

- `FunctionState.cpp::PrintAssembly(...)`

## References

[1] Poletto, M., and Sarkar, V. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS) 21*, 5 (1999), 895–913.

[2] Wimmer, C., and Franz, M. Linear scan register allocation on ssa form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization* (2010), ACM, pp. 170–179.