6.035 Code Generation Documentation

Introduction

This document details the design of a simple compiler that generates functional assembly code and the lessons we learned while building it. We did this with three passes. The first pass changed the Abstract Syntax Tree into a linked-list Mid-Level IR. Second, the Memory Manager looked at the Mid-Level IR to determine stack and registry allocations. Lastly, an ASMVisitor visited all the nodes to collect all the assembly code.

1. Work Division

We used a variety of methods to distribute the work. In the early design stages, it was important for us all to meet together to work out the design. During this initial phase, we pair programmed with Josh and David working on the Mid IR Nodes while Saif and Sajith worked on the MidSymbolTable. Later, once things were more stable, individuals worked on their own pieces, eg. Josh working on Dot visualization, Saif working on short-circuiting. Finally, during the code generation portions, meeting together was more difficult and we largely met in pairs or over remote desktop to debug the Memory Manager and Assembly test cases.

2. Clarifications, Assumption, Additions

At the high level, we assumed that we are compiling for the x64 architecture, using NASM. Since code generation was complex enough, we did little to no optimizations and only focused on correctness.

Integer overflow

In accordance with assembly operations, only the last 64-bits of an integer operation are kept. No overflow errors occur.

Runtime error handling

When a runtime error occurs, the program print an appropriate message and then exits with system code 0 instead of system code 1 due to compatibility restraints imposed by the testing mechanism.

Method Fall-off Runtime Checking

One of the semantic checks is: "If a method call is used as an expression, the method must return a result." As explained in the previous writeup, we decided to be conservative; our semantic checker fails for any non-void method that might "fall off the end". We determine if a method returns with the following logic:

1. A block with non-control structures is guaranteed to return if it has a return statement.
2. An if statement is guaranteed to return if the if-block is guaranteed to return and the else-block is guaranteed to return -- a non-existent else-block does not return.
3. For/while structures are **not** guaranteed to return due to the potential presence of breaks/continue before return statements. This analysis could have been extended to check if a return is guaranteed by ensuring that a break/continue cannot occur before the return. We omitted this analysis for simplicity.
4. Finally, a method is guaranteed to return if it has a return statement or one of its control structures is guaranteed to return.

Therefore, if we compile an assembly program, the control provably cannot fall off the end of a method, and thus, there is **no need** for a runtime check.

Division-by-Zero Runtime Checking

If the divisor of a division operation or a modulo operation is zero, the compiled program will print an error message and exit.

3. Design

Mid-Level IR

The purpose of the mid-level intermediate representation is to remove certain abstractions from the high-level IR. After semantic checking, a good deal of the structure is unnecessary and can be stripped away. The result more closely resemble assembly code, with the major exception of being hardware independent.

The focus is on flattening the code tree, eliminating complex control structures (through the use of *jump* and *label* nodes), and (naively) expanding nested expressions.

Singly-Linked List for MidIR

Since our target, assembly code, is flat in nature, we wanted out MidIR to reflect this. Therefore, we chose a singly-linked list to store all the nodes. We implemented a visitor that recursively replaces each node in the tree with a linked-list, where the head and tail of the list are linked to the previous and next siblings respectively.

Control Structures

If, for and *while* statements are the supported control statements in Decaf, and each can be expanded into a linear series of *expression evaluation*, *main block*, *jump node* where applicable. The statements of an if/else statement, then, are no longer grouped into two subtrees. The statements are in a linear linked list, with expression evaluation and jump logic inserted between.

Integer Expressions

Integer expressions are flattened through the addition of temporary variable storage. Rather than represent exprA + exprB as a tree (+ exprA exprB), the root node is replaced with recursively-expanded nodes.

```
[tempA = resultOfExprA,
tempB = resultOfExprB,
resultOfPlus = tempA+tempB].
```

The first linked node might be expanded recursively:

```
[tempAA = 1,
tempAB = 2,
tempA = tempAA * tempAB,
tempB = resultOfExprB,
resultOfPlus = tempA+tempB].
```

Each expression is decomposed into operations of at most three operands (one is the assignment operand). This also requires the design of a *namespacing* controller, able to allocate non-conflicting names and look up names given a node. Similar namespacing controllers were written for labels.

Boolean Expressions

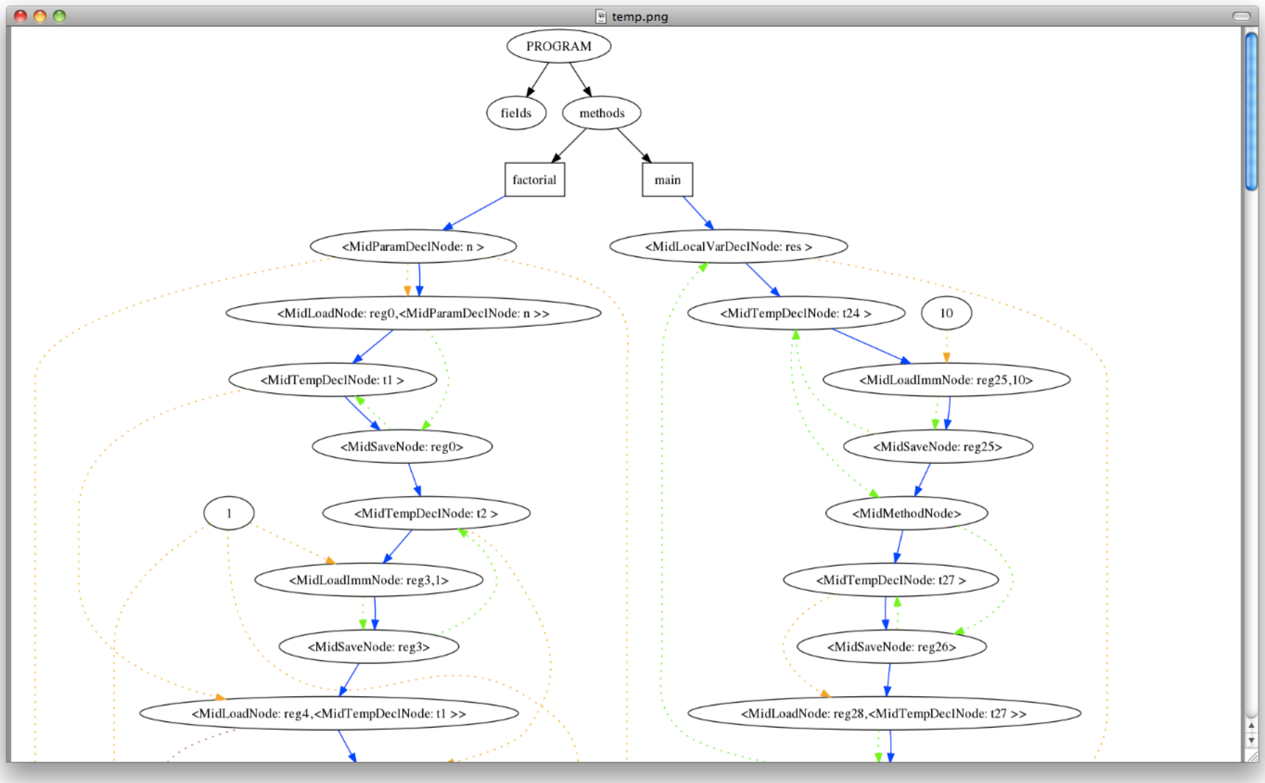
Boolean expressions are treated differently, due to the required presence of short-circuiting optimization. Utilizing the algorithm presented in class, jump nodes and block statements are assembled into a graph structure. The structure is then linearized through converting the pointers in jump statements to newly created labels.

Methods

In the mid-level IR, each method declaration remains its own linked list, referenced by a global scope object. Method calls aren't broken down yet, but rather consist of a node storing pointers to the method declaration and the nodes where parameters are assigned.

Saving Contract

As a result of only using two registers (See Memory Manager for a better description), we had each Expression node maintain a contract that their last node would be a Save Action. This allowed all the nodes to interact with each other, knowing that they can use both register and is also capable of finding the values of previous calculations. This also meant that we needed to create temporary variables when there was no explicit variable to save back to. In further revisions, we plan on replacing this non-statically-checked contract with a new datatype that enforces the contract.



A visual representation of our MidIR

Memory Manager

The purpose of our Memory Manager is to simplify the process of translating our AST to assembly. It is cognisant of the target architecture and handles register and memory allocation.

Separate Pass

Memory management was difficult enough and needed to see the full program that it made sense to run it in a separate pass

Two registers only

Due to the difficulty in register allocation, we chose to have a two-register memory model. Since each operation takes at most two registers, we move the parameters into the two registers, run the op, and then saves the result to the stack (in a temporary variable if necessary). Unfortunately, this is not statically checked, but by design, it is provably correct. There is clear room for later optimization using data-flow analysis.

Parameters automatically stored on the stack

In our design, since we have a two-egister design, we move all our parameters onto the stack for storage. To maintain consistency, even though arguments 7 and above are already on the stack, we reallocate it on the stack (with two moves to and from a temporary register).

ASMVisitor

The ASMVisitor generates a low-level IR where each line can be directly translated into Assembly. This visitor makes use of the MemoryManager to properly allocate memory. After generating the low-level IR, the visitor iterates over the results of the low-level IR to generate assembly code using the Intel x86 syntax and NASM.

Code Generation

Since assembly code is largely pattern matching and now that a node can query the Memory Manager for the location of variables and register assignments, each node is capable of generating its own assembly.

We start with boilerplate, then insert each method with a label. At the method node, it takes care of the method calling conventions and then visits all its children to collect code. A Java class was created to represent assembly operations.

Visitor Design

Throughout this document, a generic “Visitor” object for each step has been referenced. The actual visitor objects will be composed of submodules responsible for their individual tasks, coordinated by one main object.

Alternative Design

On a very high level, our design suffered from a lack of static checking. Often was the case that we didn't realize an overloaded method had *not* yet been implemented. We would only realize there was an error when a test lead to an assert false. However, this did encourage us to write additional, fairly exhaustive tests.

4. Interesting Issues

NASM ambiguity

It took us a long time to realize that a "word" refers to 16 bits in NASM, among other NASM peculiarities.

Cloud Compilation

Since two of our members are on OSX and another has a 32-bit machine, using NASM to compile and run the tests was difficult. We used an Amazon Web Service instance (AWS) to allow all of our teammates to run and test code. Public Keys were used to make the process of transferring and running code simple.
We had to modify the test script for NASM, adding in specific calls to test.sh.

Array Access

In order to dynamically access array elements, we use a register to select the correct offset in memory. Our Java method that looks up an array element therefore allocates a register. The register does not get deallocated until the array element is accessed. Therefore, it is necessary to immediately access the array element to avoid wasting one of the two registers available during method calls.

Left Associativity

We had difficulties with our arithmetic operations. When we removed left recursion, we also changed the order of associativity. For the expression 100-50-40, we first processed the right side (50-40), which was incorrect. We had to go back to ANTLR and make a slight modification to our parse tree. Using a * as a wildcard allowed us to process the arithmetic with left associativity.

Re%d Herring

Perhaps a minor detail, but we found out the hard way that “%d” in printf refers to an int type, not a “digit”. Thinking that our compiler was accidentally working with 32-bit numbers when debugging printf calls showed overflow errors, we eventually discovered that “%ld” worked properly with 64-bit integers. This was also a lesson in documentation -- Saif and Josh had discovered this earlier, forgotten to document it, and Saji later spent time debugging the same issue.

6. Known Issues

Strings are assumed to be immutable -- we place them in the .rodata section. Callouts, then, will fault if they attempt to modify the string. This is not as much of an issue as a controversial design decision, but given that Decaf programs can’t even manipulate strings themselves we decided it was a reasonable choice.

Resolved issues: "Clean" Visitor was "dirty".

In doing further tests, we realized that our Semantic Checker was faulty. Immediately before semantic analysis, we did one *sweep* to clean our tree; we replaced MINUSNodes with UnaryMinusNodes and SubtractNodes depending on the context. However, we forgot to update references to siblings when replacing nodes, which resulted in dropped siblings.