CIS*4650 Checkpoint 3 – Documentation

Overview

This document outlines what was done for the third checkpoint in the C- programming language compiler implementation. Topics include assumptions and limitations, future improvements, overall design and implementation, difficulties encountered, and significant code/design changes. In this checkpoint we build the code generator component of the C- compiler. This program takes the abstract syntax tree generated in the previous checkpoints and generates the assembly code for the input program. Once the assembly (.tm) code is created we can pass it as input for the TM simulator to execute.

Assumptions

The main assumption for this checkpoint implementation is that the program provided as input follows the C- specification. If this is true, then the compiler will be able to generate valid assembly code corresponding to the original source code.

Limitations and Possible Improvements

Besides the limitations propagated from the previous checkpoints, this code generator is limited in the sense that it can only be run on the stack-based TM simulator environment. If this condition is passed, then it can handle any valid C- construct.

Possible improvements include more comprehensive syntactic and semantic error handling (as outlined in previous checkpoints), and code optimization. In addition to adding more comprehensive semantic handling we may generate assembly code to handle semantic errors by having predictable responses to issues (such as invalid argument lists) and continuing

to execute whenever possible. To optimize the code generated by this compiler further we may optimize register allocation, remove unnecessary or redundant code, and remove tail recursion, amongst other things. As a more implementation-specific improvement one should completely separate the semantic analyzer from the rest of the program. The final compiler would then be a module composed of the many submodules.

Design Process

Given the abstract syntax tree from checkpoint one and the symbol table from checkpoint two we can navigate our program and generate the corresponding assembly code in real time. We do this first by generating a prelude set of instructions to run before we process our program. After the prelude we process the abstract syntax tree and output the code for our functions. Once the syntax tree is parsed, we can generate the final set of instructions (finale) that will ultimately call the main method in our program and clean up the corresponding activation record.

Adaptations from Checkpoint 2

To serve the new functionality of the code generator we needed to create a new visitor to traverse the syntax tree and generate the code. We also needed to copy over all semantic analyzer functionality to this new visitor. Next we needed to modify the variable declaration classes, adding fields for the nesting level and frame pointer offset. Nesting level determines which frame pointer to use (local or global) when accessing the address of a variable. Offset determines where relative to the frame pointer the address of the variable lives. To each variable we add a reference to their declaration. This access makes it easy to evaluate expressions later on. Finally, we add a new field of type Boolean to array variables. This is used to determine if the array variable is a local array or a pointer to another array.

Memory Space

Our program is organized in the simulator's "main memory" as a stack. At the top of the stack (highest memory address) is the area for global variable declarations. After this we'll have the activation records (stack frames) for each of the functions in the program. Since we only have integer types in C- we don't have to worry about different offsets for different types. As we process variable declarations, we store them in the symbol table along with their location relative to the stack frame. As we process function declarations, we store them in the symbol table with their address in the instruction space (line number). When we later evaluate expressions, we can use these relative locations to fetch the memory addresses and perform various operations. The symbol table from the previous checkpoint makes sure we are always fetching the correct symbol by enforcing scoping rules.

Instruction Space

The instruction space defines the memory location where each section of code lies (i.e. code blocks for functions). When we keep track of this, we can perform jumps to different sections of code and control the runtime flow of our program since each instruction in the TM simulator is executed in order of line number.

Implementation Process

We start by constructing a **CodeGenerator.java** class that implements the same *AbstractVisitor* interface as the *SemanticAnalyzer* in checkpoint two and *ShowTreeVisitor* in checkpoint one. This will be used to generate our code. The *Code Generator* is composed of a *Symbol Table* and *Type Checker,* the same as the *Semantic Analyzer.*

Adaptations from Checkpoint 2

There were no functional changes made to the semantic analyzer. The **AbstractVisitor.java** interface was modified to account for new functionality in checkpoint three. Although the *level* parameter in each *visit* method could be reused to instead keep track of the frame offset (stack pointer), an additional parameter *isAddress* needed to be added to help in evaluating assignment expressions. Also added was the return of an integer value. Syntax nodes will sometimes share the same scope (i.e. function params and function body) and the integer return lets us keep track of the offset updated by inner syntax nodes.

Memory Space

To make sure we are outputting instructions related to the correct memory space we make use of two offsets, one for global and one for local. As we begin traversing the syntax tree, we start with global variable declarations. As we process these variables, we decrement the global offset and add them to the symbol table with a *nestLevel* of 0. Next we process functions. Since each function has its own scope, we make use of the frame offset. Starting at -2 for each function (frame pointer – 0 is for old frame pointer, frame pointer – 1 is for return address) the stack pointer is decremented to account for memory of local variable declarations. When variables are used inside of expressions, we use the reference to the variable declarations to capture the declaration offset and nesting level. The nesting level tells us which frame pointer to use to access the variable (local or global) and the offset tells us how far down the stack to go to get it. Also, when we compute expressions, we need temporary variables to store the intermediary values. To implement this, we push intermediary values to the bottom of the stack when they're being computed then collapse the stack and store the final value.

Instruction Space

To make sure we are executing the correct instructions, we need to have a pointer keeping track of the current line number (*emitLoc)* and a pointer to mark the entry line number of our program (starting line number for main method). To ensure the correct control flow of our program we make use of values in registers and the program counter register to jump to different addresses in the instruction space. Since the TM simulator sorts instructions based on line number before loading them to be executed, we can print instructions out of order when it is convenient to do so (conditional and conditional jumps).

Lessons Learned

Locking yourself early on to a specific code structure can impact you negatively in the long run. Although the java class structures were sufficient for checkpoints one and two, some revisions needed to be made to account for checkpoint three functionality. Most notably the abstract visitor class needed to be changed drastically (return type and parameters) which meant going back and making changes to the old visitors (show tree and semantic analyzer).

Aftermath – Implementation Issues

There were several barriers to overcome for implementation. The most troubling were handling shared scope between syntax nodes, function calls, and passing arrays as arguments.

To handle shared scope between nodes such as parameters within a function declaration (*VarDecList.java)* and elements within a function body (*CompoundExp*) required the redesign of the *AbstractVisitor.java* interface mentioned above.

Handling function calls with nested expressions was particularly difficult. Initially I had tried to evaluate argument expressions in the current function scope before passing them on to

the next function as arguments. I ran into trouble with this implementation when I encountered arguments with expressions as this led to the overwriting of the new frame pointer before making the context switch. I solved this issue by instead evaluating expressions in the new function activation record.

Perhaps most difficult was the passing of arrays to other functions by reference. You must differentiate between when the array is a local reference (address points to index of first element) and when the array is a reference to a passed-in parameter (address points to the base address of the initial array). To remedy this, I modified the *ArrayDec.java* class adding a variable *isAddress* to capture whether the declaration is a pointer to the data or a pointer to the original address. Based on this value, when we are passing an array variable as an argument, we either load the argument as an address or a value in memory (also an address). When we are accessing an array index, we do a similar check to ensure we're performing the right operation.