COMPUTER SCIENCE
& ENGINEERING
TEXAS A&M UNIVERSITY

CSCE 434: Compiler Design

Fall 2023

Cocopiler

December 4, 2023

Author	*UIN*
Caleb Austin	830006168

Summary

Cocopiler is an optimizing compiler for the Coco programming language. It runs an input program through a series of levels to arrive at DLX machine code. Those levels are the scanner, AST, type checker, IR, optimizer, register allocater, and code generator.

The scanner takes the original program and turns it into a series of tokens. The AST, abstract state tree, takes those tokens and turns them into a tree. The type checker is part of the AST and checks the types of everything. The IR, intermediate representation, is a series of blocks generated by the AST that serve as an in-between between the structure of Coco and the unstructured nature of DLX. The optimizer takes in the IR and looks for ways that it can simplify it without changing what it does. The register allocater takes in the IR and allocates registers to variables. The code generator takes the IR and the register allocation and generates DLX machine code.

Said more succinctly, the scanner takes the program and grabs the meaningful parts (statements and variable name and not comments or spaces), passes it to the AST which puts those tokens into a meaningful structure.

Cocopiler is not fully complete yet, unfortunately. It is mostly functional; however, there may still be some bugs, arrays do not currently work, and block folding is not implemented.

Feature Check List

- ☐ Error Recovery
- ✗ SSA
- ✓ Type Checker
- ✓ Constant Folding
- ✓ Constant Propagation
- ✓ Copy Propagation
- ✓ Orphan Function Elimination
- ✓ Uninitialized Variables: explicit handling (required)
- ✓ Common Subexpression Elimination
- ✓ Dead Code Elimination
- ✓ Graph-coloring Register Allocation
- ✗ Register Allocation - Coalescing

KLOC

Code:	6788
Comment:	250
Total:	8203

Implementation

Cocopiler is implemented as a series of levels that progressively move the program from Coco to DLX or check or improve the program. Those levels are the scanner, AST, type checker, IR, optimizer, register allocater, and code generator.

Interface

The Compiler class provides the user with an interface with which to control the compiler and the CompilerTester class uses the Compiler class to implement a command line compiler.

Compiler(Scanner scanner, int numRegs)

The Compiler class can be created by providing the constructor with a Scanner from which to read in the tokens and the number of registers the program is meant to use. Is it up to the user of the Compiler class to create a Scanner. The Compiler can take a maximum of 24 registers. Since there is no downside to increasing the number of registers the program uses (unless you set it above 24 and break the program), it is recommended that you always set this to 24.

genAST()

The genAST method takes the scanner given in the constructor and passes it to the AST class constructor. The AST handles creating itself entirely on its own. The compiler saves the AST internally and returns it to the user.

genSSA(AST ast)

SSA is not implemented, so this method just returns the AST passed in.

optimization(List<String> optStrings, CommandLine optsCmd)

The optimization method performs optimizations on the IR to improve program performance. If the IR has not already been created the compiler will generate it. This will typically be the case since there is no genIr method. It is stored internally once it has been created. The IR is generated by the AST and the generation fully handled by the AST and IR internally. The IR is represented as a list of Graph objects, one for each function. The optStrings list is used to determine which optimizations should be run. All optimizations are run if "max" is included in optStrings. The zeroing of uninitialized variables is always done. Dead code elimination is run when when "dce" is included, constant folding when "cf" is included, constant propagation when "cp" is included, common subexpression elimination when "cse" is included, copy propagation when "cse" is included, and orphan function elimination when "ofe" is included. All of the optimizations, with the exception of the orphan function elimination, are implemented in the graph objects. The ofe relies on the graphs to provide it with a list of functions they call. The optsCmd argument is not used.

regAlloc(int numReg)

The regAlloc method generates register allocations for all the variables in the IR. Like the constructor it takes a maximum of 24 registers. It generates the IR if it does not already exist.

It generates an allocation for each function. The allocations are stored internally once they are generated.

genCode()

The genCode method runs the register allocation if it has not already been run and then generates the DLX code as an array of integers.

hasError()

Returns a boolean indicating whether or not an error has occurred.

errorReport()

Returns a short description of the error that occurred.

Scanner

The scanner is implemented in the Scanner class. The scanner takes in a Reader that should provide it with the program you want compiled. It allows the user, typically the compiler, to read in the program as tokens rather than characters which more convenient for the compiler. The scanner internally put the reader into a BufferedReader which allows me to reverse the input if necessary. When the user asks for a token the scanner reads character by character narrowing in what type of token it is reading with each character. Once it has read enough to know what kind of token it is looking at it creates a Token object and returns it. For the most part the scanner can read straight through the input without needing to back up. There are however, some token that cannot be differentiated without looking a character ahead, necessitating a reverse. The token has a line number, character position, kind, and lexeme (the string that it derives from). The kind and lexeme are used by future levels of the compiler, while the line number and character position are just used for error reporting. The scanner exposes three methods in addition to its constructor.

Scanner(Reader reader)

Sets up the scanner for use.

Error (String msg, Exception e)

This prints an error message for the user given as exception and a message string.

hasNext()

This tells the user whether there is anything left to read from the scanner.

next()

This returns the tokens to the user.

AST

The AST is implemented in the AST class. The AST is implemented in an object orientated manner; so an AST object takes in a scanner, checks for necessary tokens, uses the tokens to orientate itself, then passes the scanner to other classes which it creates objects which handle their own construction internally. The AST calls genAST on those objects once that have been constructed to create their parts of the AST. The objects are also passed the variables table when they are constructed to allow them to check for reused variable names. Unfortunately, the AST is poorly designed and there are a lot of complexities with using it.

Type Checker

The type checker is implemented in the AST and a separate TypeChecker class. Those are also some other classes in the types package that help. The TypeChecker class provides a check method that takes an AST object. The TypeChecker then calls checkType on the AST. Every node in the AST (except those that don't, my AST is weird) has a getType and checkType methods. The Type class is defined in the types package. The checkType method takes a TypeChecker, Type, and String. Each node in the AST calls checkType on its children. The getType method is used to help the checkType method. The TypeChecker has a reportError method and is passed to every node to allow them to report errors they find. The Type and String are the expected return type of a function and the name of that function. They are set by the AST (main) and FunctionDeclaration nodes. The return type is used to check for an error and the string is used to report an error if one is found.

IR

The IR is implemented primarily by three classes: Graph, Block, and Instruction. An InstructType enum is used to keep track of the types of instructions, a Variables class is used to generate temporary variables, and a ValueCode class is used in the construction of the IR. The are also a GraphIterator and a BlockIterator.

The IR is generated by calling the genIr method of the AST. It generates a graph for each function in the program and returns them in a list. It also records this, so the IR does not need to be regenerated if the method is called again. With the exception of the AST and FunctionDefinitions class, the nodes in the AST implement the genCode method. The genCode method takes in a variables object, to provide the temporary variable name, and returns a ValueCode object. The ValueCode object is exactly what it sounds like. An object containing a value and some code. The value is a temporary (or null) containing the result of the previous calculation. The code is a list of instruction objects. This design allows me to derive an IR from my object orientated AST, since a node generating some instructions does not need to know what will be done with the result. For example, a Operation node can create an instruction for 2 + 4, store the result into a temporary, make that temporary the result, and not need to worry about whether that temporary will be used in another calculation, passed to a function, etcetera.

The Graph objects take in a ValueCode object along with some information about the function it implements and the set of global variables. The Graph extracts the the instructions from the ValueCode object and breaks it into blocks.

Optimization

The optimizer implements seven optimizations: zero uninitialized variables, dead code elimination, constant folding, constant propagation, common subexpression elimination, copy propagation, and orphan function elimination.

To implement these seven optimizations, it uses three analyses: live variables analysis, available expressions analysis, and set variables analysis.

The live variables analysis supports the dead code elimination and the register allocation.

The available expressions analysis supports the constant propagation, common subexpressions elimination, copy propagation.

The set variables analysis supports the zeroing of uninitialized variables.

For dead code elimination, constant folding, constant propagation, common subexpression elimination, and copy propagation, the optimization method in Compiler calls a function in the Graph, that function runs the analysis until it is no longer changing and then runs the optimization until it is no longer changing. The analysis is rerun to convergence for each run of the optimization. The Graph calls the blocks to run the optimizations and the analyses. (Constant folding does have an analysis.)

Zeroing uninitialized variables working similarly, expect is only needs to run once and it needs to run before the other optimizations (otherwise they miss out on chances for optimization).

Orphan function elimination is different. It runs in the Compiler. It starts with main, it adds it to a list and a queue. The queue is it popped from until it is empty. For each function (Graph) that is popped out of the queue, all of the functions they call are added to the queue (unless they were already in the queue). All the functions seen through out this are put into a list, then any function not in that list is removed.

Register Allocation

The register allocator can allocate a maximum of 24 registers. It is recommended that you use all of them unless you have some good reason not to. 24 is the maximum, because 0, 28, 29, 30, and 31 are reserved for special purposes in DLX and I use 25, 26, and 27 for special purposes in my code generation. That leaves 1-24 available for use. I generate a separate register allocation for each function (Graph).

To allocation the registers I run the live variables analysis to convergence and go through each block. At each block I generate the live sets, this is done by the blocks. I create LiveRange objects for every variables I see. The LiveRange objects include the name of the variable, where that variable is live. After looping through the blocks, each live range is given the list of live ranges so it can find the live ranges it interferes with and is given a reference to the stack so that it can give an accurate count of the live ranges it interferes with that are not in the stack. To build the stack of live ranges, the list of live ranges is sorted from fewest to most conflicts. The live ranges are removed in order of the number of conflicts that have until all are in the stack. They are subsequently popped out of the stack and given register allocations. The allocations are determined by the live range, which picks the lowest available register.

Code Generation

The code generation is implemented in the Compiler class.

The first things it does is reserve space in memory for every global variable. Next, it generates code every function. The compiler does not yet know where the code for each function starts, so a map is used to get keep track of where all the function calls are. In the main function the frame register is set at the beginning. Subsequently, it is set before the function call. A space is then allocated for every variable. (This is one source of inefficiency, since space is reserved for every register, even if no variable is in it.) Next, the stack register is set. The code generator uses a VariableLoader class to help it load and unload variables, a different object is used for each function. Then, the function parameters and global variables are loaded into registers to make them available.

Finally, we begin looping through the blocks and instructions. I use a Code class to put all the values in the right stop. I also have an Op enum for the DLX operations I use. The ops have their format and opcode embedded inside themselves for convenience. Jumps are implemented using the conditional branching operations, even if there are unconditional, to avoid changing the return value in register 31. Like with function calls, some jump instructions need to be put in a list and revisited later since I do not yet know where those operations occur. After the code is generated in a function there is nothing left to do but store the code.

Once I have looped through all the functions, I pick a start location for each function, making sure the main function is first, then I fill in the jump locations for the calls, and turn each code object into an integer in an array.

Stack Setup

Per convention, I store the frame pointer is register 28, stack pointer in 29, global pointer in 30, and return pointer in 31.

The global pointer points to the global variable space with doesn't move, so the pointer doesn't change.

The frame pointer pointer one word above the function variables storage space. (Remember that the stack grows downward, so one above means one behind.)

The stack pointer pointer to the word just after the function variables storage space.

When a function is called its parameters are stored in memory just below the function variables and the current return value is placed immediately after the parameters.

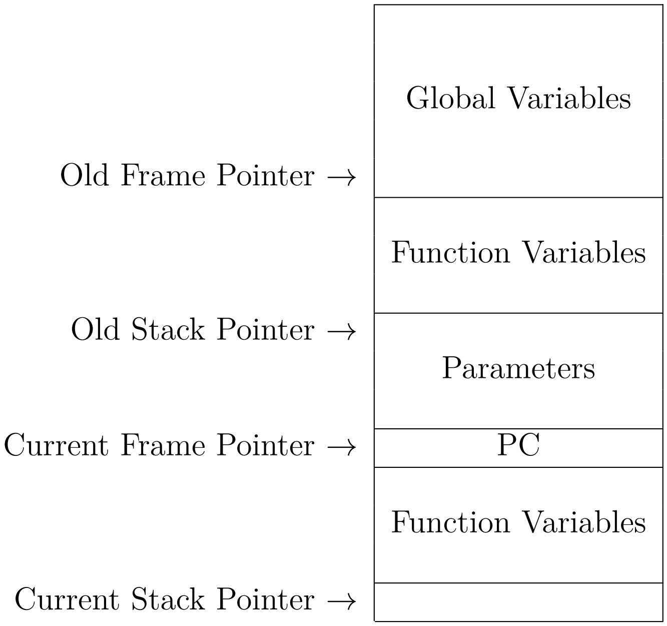

Figure 1: Shows stack setup after a single function call.

Updates after Evaluation

Since the evaluation, I have made a lot of progress on improving the optimizations and have fixed an array of other problems. I have changed how optimizations exempt global variables to improve them. I have allowed constant folding to fold when only one of the operands is know if it has trivial result. I got rid of the ghost block which didn't appear in the dot graphs, but prevented some optimizations. I fixed function overloading. Removed copies between the same register from the code.

Reflection

What went well

I feel overall I had a solid grasp the concepts that go into building a compiler through the project. I feel that there were a few solutions I came up with that were quite elegant and worked really well, such as the Code class and the ValueCode class.

What did not go well

There were some design decision that I made in this project that turned out to cause a lot of problems later on. The first was my decision to fully object orientate my AST. While that did occasionally allow for some very elegant code, it also forced a lot of code duplication and created ugly code in places where I had to handle irregularities in the AST. I also feel that it was a design blunder to use strings for all the variables and values in the IR, since it forced me to do a lot of repeated string manipulations.

Advice

I feel some of the biggest design flaws in my project stem from my desire to do things my own way rather than just following what Professor Nguyen gave in the starter code. However, I feel I learned more because of that. I would advise that using the starter code will make your work easier, but it may not learn as much. Other than that you will likely benefit a lot if you read the textbook and try to start working on things early.

Solo version

I affirm that this report describes the correct status of my submission as of December 4, 2023.

{Caleb Austin}