Project 3
Decaf Code Generation

Premature Optimization
Manuel Cabral
Jason Paller-Rzepka
David Stingley

Objective

Our compiler takes in code, parses it, then converts it to Abstract Syntax Trees (ASTs) and Symbol Tables, and then runs semantic checks on it. For this deliverable, our compiler must convert the program into assembly code. Instead of directly converting an AST to assembly code, the compiler should convert the ASTs to another intermediary representation, Control Flow Graphs (CFGs), which can then be used to generate the assembly code. It is desirable to use CFGs, because they are easier to optimize than implementations that directly convert AST to assembly, which will be helpful in later deliverables.

Data Structures

The two data structures of interest are Abstract Syntax Trees and Control Flow Graphs. They are both intermediary representations. The former is useful for semantic checks, while the latter is more conducive to generating assembly code.

The Symbol Tables and the Abstract Syntax Trees (ASTs)

There are different kinds of Symbol Tables. At the top level there is a single table that points to callouts, global fields, and the method symbol table. Inside of the method symbol table, there is a list of methods. Each method contains its own AST, which is a simplification of the parse tree that retains the necessary information to describe the program.

Program

- String **programName** - name of the program
- NodeSequence<Callout> **callouts** - list of the callouts
- Scope **globals** - list of global fields
- NodeSequence<Method> **methods** - list of methods

A program is made up of three major sections: callouts, global fields, and methods. This data structure reflects that by storing that information in three different sections.

Callout

- String **name** - name of the callout

A callout is an imported function. The compiler only needs to know the name of the function and trusts that the Decaf program uses that callout with the correct arguments.

Method

- String **name** - name of the method
- ReturnType **returnType** - the expected return type of this method

- ParameterScope **parameters** - contains the parameter variables of this method
- Block **body** - contains the immediate local variables and the statements of this method

A method is composed of its signature (return type and parameters), local variables, and statements. Due to Decaf's syntax, the immediate local variables can be separated from the statements.

Block

- Scope **scope** - contains the variables in the immediate scope of this block
- NodeSequence<Statement> **statements** - list of statements for this block

Block contains information about the variables which can be accessed at this scope, as well as the list of statements that it runs. Methods, if statements, for loops, and while loops all contain blocks.

Scope

- List<FieldDescriptor> **variables** - contains all the variables declared in this scope.
- Optional<Scope> **parent** - the Scope above this.
    - The parent of a *local* scope is either a local scope or a parameter scope.
    - The parent of a *parameter* scope is a global scope.
    - A *global* scope doesn't have a parent.

Scope only contains information about the variables inside of it and the parent scope. Since if statements, for loops, and while loops contain their own scopes, the parent scope of any of those three must also be a local scope, which will either be a method scope or a if/for/while scope. Scopes can determine from its variables how much space it needs.

> Design Decision: Allocating Space At the method level, we choose to allocate the sum of the memory required for the immediate variables of a scope and the maximum of the memory required for all the statements of the block corresponding to this scope. This makes it easier to guarantee that there is enough space for the method.

Control Flow Graphs (CFGs)

Control Flow Graphs are made up of Control Flow Nodes. There are only two kinds of Control Flow Nodes, sequential and branching. To aid the construction of these nodes, our compiler uses two different kinds of graphs, Bi-Terminal Graphs and Control Terminal Graphs.

> Design Decision: Labeling Nodes We chose to make a unique id for all nodes in the control flow graph. This will allow us to uniquely identify each node with a hashcode, and thus generate unique labels for each node easily.

Sequential Control Flow Nodes

- Optional<Instruction> **instruction** - the instruction associated with this node. If this is not present, then this node has no operation and is intended for ease of construction of the control flow graph
- Optional<ControlFlowNode> **next** - the next control flow node. If not present, then this is the end of this graph
- String **name** - the name of the node. This is useful for determining the purpose of nodes with no operations

Sequential control flow nodes go to only one node and they may or may not have an instruction. To aid in understanding what the node represents, the node has a name.

Branching Control Flow Nodes

- JumpType **type** - denotes which kind of conditional jump it should use, for example JNE
- ControlFlowNode **trueBranch** - the node to go to if the conditional is true
- ControlFlowNode **falseBranch** - the node to go to if the conditional is false

Branching control flow nodes are the only kind of node that may go to two different nodes. These are only used for the conditionals in if statements, while loops, and for loops.

Groupings of Control Flow Nodes

There are two main abstractions used to represent groupings of nodes: Bi-Terminal Graphs and Control Terminal Graphs. Bi-Terminal Graphs represent a self-contained series of nodes. In this case, only the beginning and end need to be exposed. Control Terminal Graphs deal with graphs that may not have dealt with break, continue, or return statements yet, and so expose nodes that need to be connected at a higher level.

Bi-Terminal Graphs

- ControlFlowNode **beginning** - the start node for this graph
- SequentialControlFlowNode **end** - the end node for this graph

This graph represents a collection of sequential nodes that have been grouped together as one graph. It provides access to the the first and last node.

Control Terminal Graphs

- ControlFlowNode **beginning** - the entry node for this graph
- SequentialControlFlowNode **end** - for regular execution, the exit node for this graph
- ControlNodes **controlNodes** - where control should go if break, continue, or return are used.
    - SequentialControlFlowNode **breakNode** - where escaped break statements go

- SequentialControlFlowNode **continueNode** - where escaped continue statements go
- SequentialControlFlowNode **returnNode** - where escaped return statements go

Control Terminal Graphs are used to handle escaped break, continue, and return statements. They expose nodes that represent escaped control flow statements. For example, if there is a nested if statement inside a while loop, then any break or continues in the if statement should be handled at the while level, while any returns should be handled at the method level. See figure below for overview.

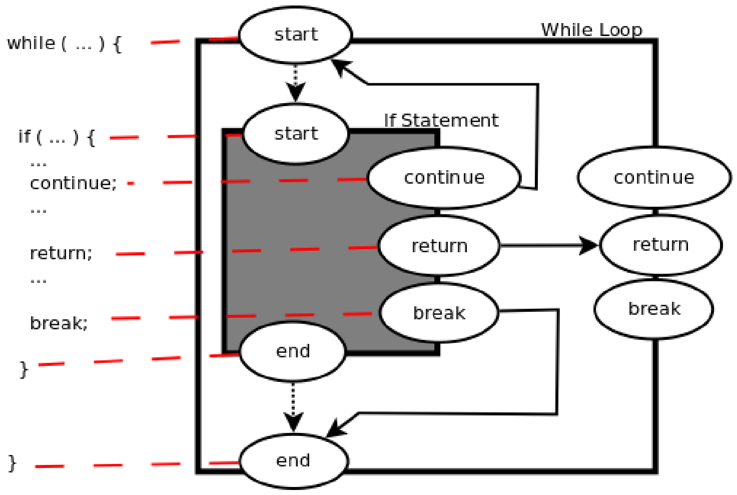


Figure: Shows how the while loop hooks up the continue, return, and break nodes

Converting ASTs to CFGs

Each AST represents a different method in the Decaf program. The compiler generates a CFG for each method.

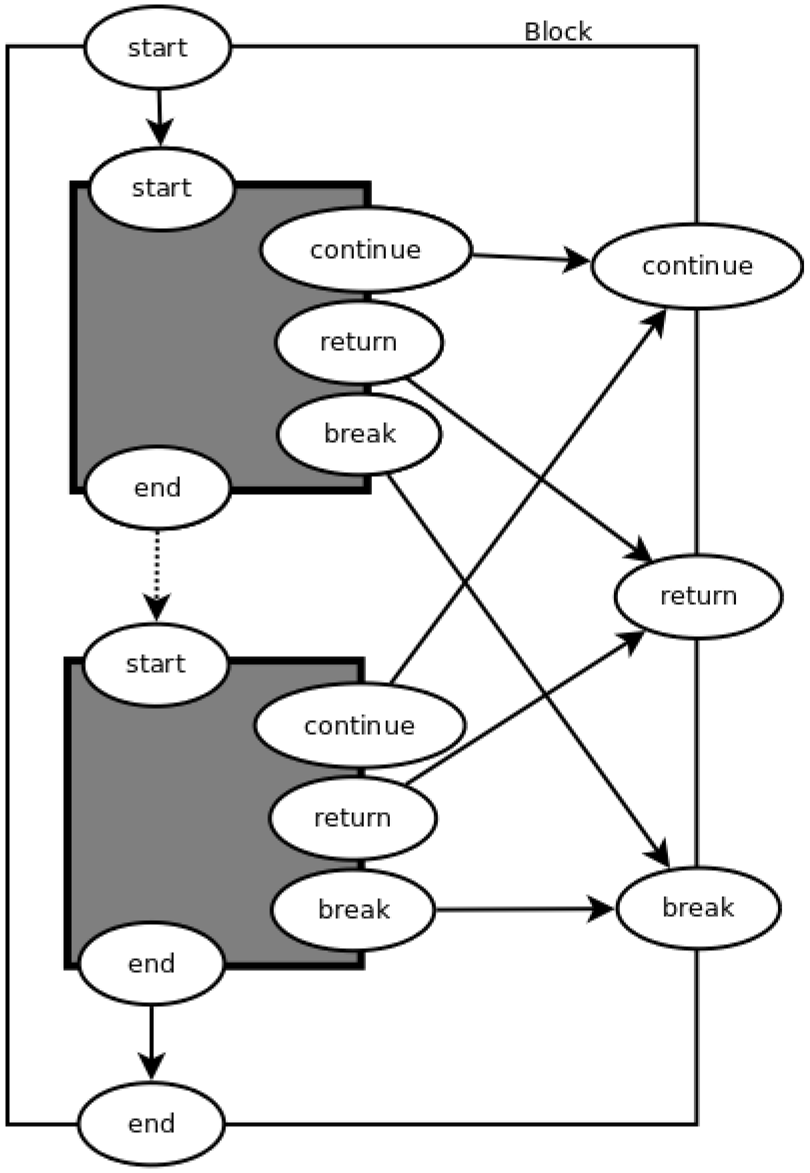
Design Decision: Mutability In previous steps of the compiler, we chose to enforce immutability of all our objects. Nodes in the control flow graph need to know where to go to next. This means that if we wanted to preserve immutability, we would have to build our graph in reverse, which would've made the code more complicated. We opted for simplicity.
Design Decision: Register Allocation We have chosen to primarily use R10 and R11 whenever possible. The only instances when this is not true is when a method is called, in which case, the arguments are put in the registers according to x86 calling convention and the return value is put in RAX. While this significantly slows down the performance of the assembly program, it significantly simplifies the code generation.

Method

Generate the CFG for the Block that corresponds to this method. Makes sure that there is an *enter* instruction that allocates the appropriate amount of space before the graph. Also makes sure that there are *leave* and *ret* instructions after the graph.

Block

A block is a collection of statements. The CFG generator for blocks creates a Control Terminal Graph, where all break, continue, and return nodes from statements are connected to the exposed break, continue, return nodes. The statements themselves are converted to Control Terminal Graphs which are connected appropriately by the block generator.



Statements

Assignment

The assignment is broken down into two different classifications: 1) Is the location that the value will be stored in a scalar or an array value? 2) What is the assignment operator?

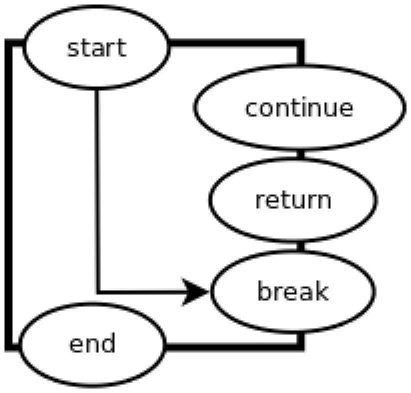
This is how the location type is handled:

1. If the location is a scalar, there are three cases:
    a. If the scalar is a global field,
    b. If the scalar is a parameter field, then stores the
    c. If the scalar is a local field,
2. If the location is an array location,

The assignment operators are -=, +=, and =. The last one just directly assigns the value to the place on the stack implied by the location. The first two retrieve the current value in that location, apply the correct arithmetic operation, and then store the result in that location.

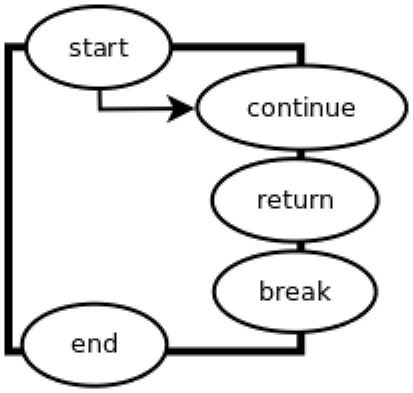
Break Statement

Generates a Control Terminal Graph, which hooks the start node directly to the break node.



Continue Statement

Generates a Control Terminal Graph, which hooks the start node directly to the continue node.



For Loop

The CFG generator for For Loops is similar to the Block's generator, except that it connects the continue and break differently. Continue statements go to the beginning of the loop, and break statements go to the end. Returns also make sure to clean up the for loop before exiting the graph.

It also has added logic for the looping variable. The CFG checks that the looping variable is less than the upper limit. If the check passes, then it executes the block, updates the looping variable, and loops back to the beginning of the for loop. Otherwise, it cleans up and exits.

If Statement

Generates a Control Terminal Graph, which hooks the continue, break, and return nodes of its statements to its exposed nodes. The general idea is to evaluate the conditional and execute the block only if the conditional is true. If false, go to the same code that the end of the block goes to.

To evaluate the conditional, the Control Terminal Graph first generates the graph for the Native Expression, from which the value of the Native Expression is obtained. This doesn't set the comparison flags, so the value is then compared to the literal for true. Finally, the Control Flow Graph can now branch appropriately to either the beginning or the end of the Block of the If Statement.

Method Call Statement

See Method Call in the Native Expression section. This is similar, except that it makes sure to get rid of the return value if there is one.

Return Statement

If there isn't a return value specified, then the start node simple goes to the return node. See the Break or Continue Statements for how this would look. If there is a Native Expression specified, then the Native Expression is evaluated and moved to RAX, as specified by Assembly convention, and returned.

While Loop

The CFG generator for While Loops is similar to the Block's generator, except that it connects the continue and break differently. Continue statements go to the beginning of the loop, and break statements go to the end. Returns also make sure to clean up the while loop before exiting the graph.

It also has added logic for evaluating the max iterations limit, the conditional statement and looping. The CFG first handles the max iterations limit, then evaluates the conditional. If either of those fail, then it goes to the end of the while loop, where it cleans up and exits as expected. Otherwise, it executes the block, updates the number of iterations and the conditional and loops back to the beginning.

Design Decision: Handling Max Repetition: We chose to always have nodes for the max iterations even if there isn't a limit on the number of iterations specified for the while loop. We handle the cases where there isn't a limit specified by having the assembly act as if the limit check always passes. We make sure to get rid of the value on the stack that keeps track of the number of iterations always. This will also be useful in case we ever want to set a global limit on the number of iterations a while loop should have.

Native Expressions

Binary Operation of two Native Expressions

There are three cases:

1. For arithmetic operations, the compiler generates a CFG that performs that arithmetic operation on the two integer values produced by the two Native Expressions.
    a. add, subtraction, multiplication: uses the *add*, *sub*, or *imul* instruction respectively.
    b. division, modulo: both use *idiv*. This instruction sets the value of the division and the remainder in two separate registers. The compiler makes sure that the CFG returns the correct of the two values.
2. For logic operations, the compiler generates a short circuiting control flow graph.
3. For comparison functions, the generated CFG does a flagged compare of the integer values of the two Native Expressions. Based on the flags set, the CFG sets the appropriate boolean literal.

Design Decision: Signed or Unsigned There is no case in the Decaf language where we will want to perform an unsigned multiplication or division, since all integers are signed. We have deprecated unsigned multiplication and division.

Method Call

The main complexity of a method call is to make sure that the arguments are in the correct places. The compiler ensures that the CFG places the correct arguments on the registers and stack, calls the function, and pushes the return value on the stack.

Design Decision: Placement of Arguments We chose to follow the x86 convention for placing function arguments for our functions too. This allows us to use the same code to run callouts.

Ternary Operation

Generates a CFG that evaluates the conditional Native Expression, and returns the value of the Native Expression on the branch that corresponds to the value of the conditional.

Unary Operation of a Native Expression

There are three cases:

1. For array length, the compiler takes advantage of the Decaf grammar: Array lengths must be specified by an integer literal. This means that the compiler can obtain the length of an array directly from the FieldDescriptor for that array in the AST, and create an instruction to push that value directly whenever the length
2. For negation of an integer, the compiler creates an instruction to do a signed multiply of -1 on the value of the Native Expression.
3. For negation of a boolean, the compiler creates a *not* instruction directly for the value of the Native Expression

Location

The location of the variable is determined by what scope it is in. The generated graph pushes the value in that location onto the stack. There are three cases:

1. Global variables are placed in the *.bss* section of the assembly code. They can be obtained by directly calling them.
2. Parameters are obtained according to the calling convention.
3. For Local variables, the compiler recurses up the local Scopes until it finds what local Scope the variable is in. As it is recursing, it keeps track of a sum of the memory allocated for each of traversed scopes. In the scope that contains that variable, It retrieves where in that variable is. It uses the sum and the location in the Scope to generate where that variable is.

   Design Decision: Loops operate of the same space in the stack We choose to make loops use the same section in the stack, instead of allocating memory for each iteration. This ensures that we only have to clean up once, and simplifies retrieval of the local variables.

Native Literal

Native literals are essentially constants (boolean, char, or int). The graph for this has a single Instruction that pushes whatever the native literal is onto the stack.

Generating Assembly Code

The compilers uses the CFGs of each method to generate the assembly code, a list of string literals, and a list of global fields. It makes sure that the main function is denoted by the *.globl main* label.

Methods

The assembly writer generates a label and the assembly for each method in the program. Since the method names are unique, those are used as the labels of the method. It traverses the top node in a depth-first fashion. Note that because it only accesses nodes that are accessible from the start node, any unreachable nodes that were generated in the construction of the CFG are automatically disregarded at this step.

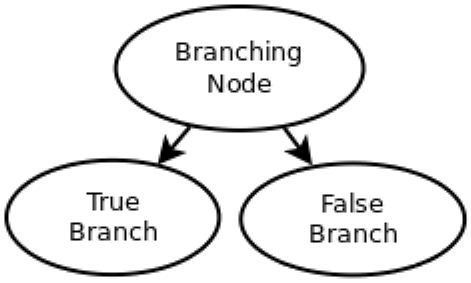
Printing out Simple Nodes

Simple nodes have at most one node that flows into it and at most one child node. For these nodes, the compiler prints out the instruction of the node, if it has one. If it does not have one, the compiler moves on to the child node. If it does not have a child node, then the compiler is at the end of this branch.

Linearizing CFGs (Dealing with Nodes that aren't Simple)

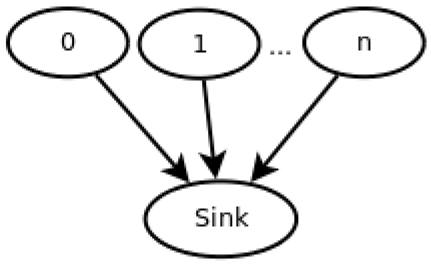
The CFG is useful for seeing control flow, but since assembly is written linearly, line by line, the compiler must now linearize the graph. To this end, the compiler makes use of jump statements to move the instruction pointer to the correct location. There are two main cases to handle:

1. Branching nodes for conditionals



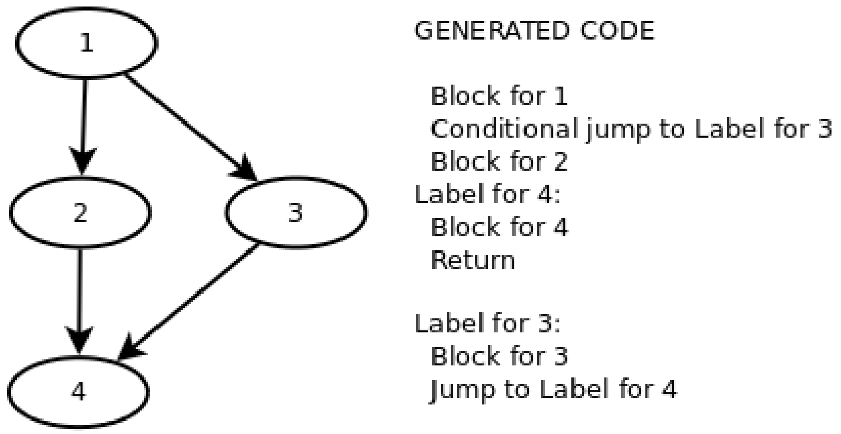
   In this case, a label is generated for the false branch, and the compiler inserts a conditional jump to the false branch. The generated code falls through to the true branch.

2. Nodes that act as a destination for multiple other nodes



In this case, a label is generated for the Sink node, and all nodes that don’t directly fall through to the Sink node jump to the label for that node.

<u>Example</u>



```
GENERATED CODE

  Block for 1
  Conditional jump to Label for 3
  Block for 2
Label for 4:
  Block for 4
  Return

Label for 3:
  Block for 3
  Jump to Label for 4
```

Figure: Block 3 is the false branch of a conditional, so a label is generated for it and it is conditionally jumped to. Block 4 has multiple blocks flow to it, so a label is generated for it. Since Block 3 doesn’t fall through to Block 4, it jumps to Block 4 after it is done.

String Literals

When generating the ASTs for the program, the compiler keeps references to all the string literals in the program. It then generates labels for each string and associates those labels to the string in the assembly code.

If the Decaf program has only has strings in the following lines:

printf(“bar”);
printf(“hello”);

Then this will show up in the assembly as:

.s_0: .asciz "bar"
.s_1: .asciz "hello"

<u>Design Decision: String Literals</u> We chose to explicitly keep track of all the string literals at the AST generation level in order to process strings more easily at the code generation level.

Globals

Globals are stored in list at the top level. The assembly code simply translates the global variables in Decaf into global variables in assembly.

If the Decaf program has the following globals:

```
int i;
boolean love[5];
boolean six[9];
int o[2];
boolean three;
int five;
```

Then these globals will show up in assembly as:

```
.g_i: .quad 1
.g_love: .quad 5
.g_six: .quad 9
.g_o: .quad 2
.g_three: .quad 1
.g_five: .quad 1
```

Design Decision: Specifying globals We chose to specify them as arrays in the *.bss* section in order to be able to print arrays and scalars in the same location for ease. This allows us to handle all the globals in a single pass.

Unresolved issues

The compiler currently has problems with array locations.