

Programming Project 4: TAC Generation

Due: Friday, August 15th at 4:00 p.m.

The Goal

In the final project, you will implement a back end for your compiler that will generate code to be executed on the SPIM simulator. Finally, you get to the true product of all your labor – running Decaf programs!

This pass of your compiler will traverse the abstract syntax tree, stringing together the appropriate TAC instructions for each subtree – to assign a variable, call a function, or whatever is needed. Those TAC instructions are then translated into MIPS assembly via a translator class we provide that deals with the more grungy details of the machine code. Your finished compiler will do code generation for all of the Decaf language (with a few minor omissions) as well as reporting link and run-time errors.

Students generally find PP4 to be close to PP3 in terms of difficulty and time consumption. The debugging can be intense since you may need to examine the MIPS assembly to sort out the errors. By the time you're done, you'll have a pretty thorough understanding of the runtime environment for Decaf programs and will even gain a little reading familiarity with MIPS assembly.

I think this is definitely the most fun of the projects; it's an awesome feeling when you finally get to the stage where you can compile Decaf programs and execute them. Try not to get too distracted playing Blackjack and ChessMeister!

Starter Files

We made some minor changes to the previously provided code. If you checked out PP3 from our `svn` repository, then you can update to the PP4 code with the following command from `myth` from the root of your checked out PP3 code:

svn update

Or checkout from scratch: `svn co file:///usr/class/cs143/assignments/svnrepos/pp_base`

Or remotely: `svn co svn+ssh://USER@myth.stanford.edu/usr/class/cs143/assignments/svnrepos/pp_base`

Or just copy the files: `cp -r /usr/class/cs143/assignments/pp4_base .`

Here is a list of the new files:

<code>codegen.h/.cpp</code>	CodeGenerator class
<code>tac.h/.cpp</code>	Tac class and subclasses
<code>mips.h/.cpp</code>	our provided TAC -> MIPS translator

The output is MIPS assembly that can be executed on the SPIM simulator. The `spim` executable is in `/usr/class/cs143/bin`. There is also an `xspim` visual version. You can

either invoke using the full path or add our bin directory to your path to use the short name. The `-file` argument to either allows you to specify a file of MIPS assembly to execute.

For your convenience, we provide a `run` script that will do the steps in sequence. You invoke it with one argument, the path to the Decaf input file:

```
-- dcc < samples/t1.decaf >tmp.asm
-- spim -file tmp.asm

SPIM Version 6.3a of January 14, 2001
Copyright 1990-2000 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/class/cs143/lib/trap.handler
hello world
```

Carefully read all the files we give you. This is particularly important here since there is a chunk of new code in the project. A few notes on the starter files:

- You are given the same parse tree node classes as before. Your job is to add `Emit` instructions to the nodes, implemented using the same virtual method business used in previous programming projects. You can decide how much of your PP3 semantic analysis you want to incorporate into your pp4 project. We will not test on Decaf programs with semantic errors, so you are free to disable or remove your semantic checks so as to allow you to concentrate on your new task.
- You are to replace `parser.y` with your own parser. You should not need to make changes to the parser or rearrange the grammar, but you can if you like.
- We have removed doubles from the types of Decaf for this project. In the generated code, we treat booleans just like ordinary 4-byte integers, which evaluate to 0 or not 0. The only strings are string constants, which will be referred to via pointers. Arrays and objects (variables of class type) are also implemented as pointers. That means all variables/parameters are 4 bytes in size—that simplifies calculating offsets and sizes.
- Interfaces are removed for code generation to simplify management of the vtable and dynamic dispatch. You are not required to generate code for method calls on objects upcasted to interface types.
- Implementing `GetByte` and `SetByte` is optional.
- The `CodeGenerator` class has a variety of methods that can be called to create TAC instructions and append them to the list so far. Each instruction is an object of one of the instruction subclasses declared in `tac.h`. The `CodeGenerator` has support for the basic instructions, but you will need to augment it to generate instruction sequences for the fancier operations (array indexing, dynamic dispatch, etc.)
- The `Mips` class, which we provide, is responsible for converting the list of TAC instruction objects as part of final code generation. This class encapsulates the details of the machine registers and instruction set and can translate each

instruction into its MIPS equivalent. You will not likely need to make changes to this class.

CodeGenerator Implementation

As a suggested checkpoint, you need to implement code generation for a single main function without arrays or objects. Here is a quick sketch of an order of attack:

- Before you begin, go back over the TAC instructions and examples in the TAC Handout to ensure you have a good grasp on TAC. Also read the comments in the starter files to familiarize yourself with the `CodeGenerator`, `Tac`, and `Mips` classes we provide.
- Plot out your strategy for assigning locations to variables. A `Location` object is used to identify a variable's runtime memory location, i.e., which segment (stack vs. global) and the offset relative to the segment base. Every variable, be it global or local, a parameter or a temporary, will need to have an assigned location. Figure out how/when you will make the assignment. As a first step, you may want to print out each location before doing any code generation and verify all is well. If you aren't sure you have the correct locations before you move on to generating code, you're setting yourself up for trouble. Once you have assigned locations for all variables located within the stack frame, you can calculate the frame size for the entire function, and backpatch that size into `BeginFunc` instruction.
- The label for the `main` function has be exactly the string "main" in order for `spim` to start execution properly.
- Start by generating code to load constants and add support for the built-in `Print` so you can verify you've got this part working. Simple variables and assignment make a good next step.
- Generate instructions for arithmetic, relational, and logical operators. Note that TAC only has a limited number of operators, so you must simulate the others from the available primitives. In the past, students seem tempted toward complex implementations involving strange branches and `ifz/goto`, but there are simple, straightforward solutions if you think carefully about it.
- Generating code for the control structures (`if/while/for`) will teach you about labels and branches. Correct use of the `break` statement should work for exiting while and for loops.
- Take note of what Decaf built-ins are available and how each is used. A few trouble spots in the past for students have been making sure Booleans print as `true/false` (not 0/1) and that `==` on strings compares the characters for equality, not just the pointers.

At this point, you should be able to handle any sequence of statements in a single `main` function (not including arrays and objects). Try finishing everything up to this point by Wednesday, especially if you're out of late days. Going on, you will finish the rest of code generation, which includes these tasks:

Going on you will finish the rest of code generation, which includes these tasks:

- To handle multiple functions, you now need to assign locations to the function parameters and figure out your strategy for assigning function labels. We suggest using the function name prefixed with an underscore as the function label and for classes to further prefix with the class name. You're welcome to use any scheme you like as long as it works (i.e., assigns unique labels with no confusion).
- Plan your array layout. Remember that your generated code is responsible for tracking the array length. Where will you store that? When is the length set? How do you access it? Once you have a strategy, implement the built-in `NewArray` and the `length()` accessor. Add a runtime check that rejects an attempt to create an array without a positive number of elements, printing the message

```
Decaf runtime error: Array size is <= 0
```

and halting execution. (Templates for the error messages are provided in `errors.h`)

- Code generation for array elements requires computing offsets and dereferencing. Be careful to consider both the case when the array element is being read and the case when it is being written. Include a runtime check for array subscripting that verifies that the index is in bounds for the array. If an attempt is made to access an out-of-bounds element, at runtime you should print the message

```
Decaf runtime error: Array subscript out of bounds
```

and halt execution.

- Now consider how you will configure objects in memory— in particular, think through how you will access instance variables and implement dynamic dispatch. Sketch some pictures and be sure to consider how inheritance will be supported. With your plan in hand, figure out how you will assign locations to instance variables and methods. Add code to generate the class vtable.
- Add implementation for the `New` built-in, taking care to generate the necessary code to set up the new objects' vtable pointer.
- Code generation for instance variable access is somewhat similar to array element access in that it involves loads and stores with offsets. It might help to suspend semantic processing while testing (i.e. act as though all object fields are public) so that you can directly read and write the fields of an object from the main function.
- Method calls are handled similarly to function calls, but dynamic dispatch and the hidden receiver argument adds some complication. Refer back to the earlier object pictures on the lecture slides to ensure you understand what code must be generated to jump to the correct method implementation. Remember that there is an additional argument "this" that needs to be passed as a behind-the-scenes parameter when generating code for a method call. In the context of a method body, you will

need to synthesize a location for the identifier "this" at the offset for where the parameter can be found.

- You should add one piece of "linker"-like functionality to verify that there is a definition for the global function `main`. The error reported when the program contains no main is:

```
*** Linker: function 'main' not defined
```

If there is a link error, no code should be emitted.

Hints and suggestions

Just a few details that didn't fit anywhere else:

- The switch `tac` can be used to skip final MIPS code generation and just print the TAC instructions. This is quite useful when you are developing. Note that it is **not** expected that your instruction sequence exactly match ours. Depending on your strategy, you can get many functionally equivalent results from different sequences. We will not be using `diff` on the TAC or MIPS sequences, only on the `spim` output.
- There are links to documentation and resources for the SPIM simulator on the right-hand side of the CS143 website.
- We included comments in the header files to give an overview of the functionality in our provided classes but if you find that you need to know more details, don't be shy about opening up the `.cc` file and reading through the implementation to figure it out. You can learn a lot by just tracing through the code, but if you can't seem to make sense of it on your own, you can send us email or come to office hours.

Testing

There are various test files that are provided for your testing. For each Decaf input test file, we also have provided the output from executing that program's object code under `spim`. There are many different correct ways of sequencing the instructions, so it's not helpful to compare TAC/MIPS outputs, but the runtime output should match. Be sure to test your program thoroughly, which will involve making up your own additional tests.

We will test your compiler only on syntactically and semantically valid input. We will expect your code to report errors *only* for the runtime and linking errors specified above.

We provide a "beta" version of the usual testing framework for `samples-pp4` (e.g. it is new, cool, and more untested than we'd like). In addition to these tests, we also provide some examples of TAC code generation in `samples-tac` and some examples of programs written in Decaf by past students in `samples-students`.

Optional extensions

Now that you have a working compiler, the range of fun extensions is wide open. Perfect for absorbing all that unpleasant extra time you have on your hands because you saved all of your late days!

- If you look at the printed sequence of TAC instructions, you will find much obvious inefficiency that cries out for attention. There is a lot of opportunity to reduce the number of instructions by streamlining the code, combining steps, eliminating redundancies, and the like by applying local optimizations such as constant folding, constant and/or copy propagation, algebraic identities, strength reduction, temp re-use, common subexpression elimination, and others. For those of you with more background or interest in architecture/assembly, feel free to poke around in our `mips` class and experiment with peephole optimizations or work on improving its register allocation strategy. If you like optimization, you will love CS243!
- Can you add the `switch` statement or other constructs from your PP2 extensions?
- Supporting interfaces, including dispatch on an object upcasted to an interface, is an excellent task for learning about the implementation cost for this feature. There are several implementation strategies you could use for this, you may want to research the typical behavior for Java interfaces and C++ multiple inheritance as a starting point.
- Change the code generation to do short-circuit evaluation for logical operators.
- Add a pass-by-reference parameter mechanism to Decaf.
- What about adding function pointers, inlining, or variable or default arguments?
- Write a garbage collector. Better yet, make your Decaf-to-C++ output do it too.
- Make your compiler into a debugger too which lets you step through code like `gdb`.
- Submit an interesting program written in Decaf.

Whether or not you decide to add some bells and whistles, please make sure that what you submit will work with our scripts on the base requirements. You can include an alternate version with your submission or have a means of enabling special features with flags, just explain in your `README` file what we need to do to try it out. Be sure to include some sample files with your submission that show your extra features in action and tell us about those files in your `README`.

Grading

Most of the grade will be allocated for correctness (85%). There will also be some consideration (15%) for design and implementation (see the “Coding Guidelines” page on the website). We will run your program with the given tests as well as others of our own. We will use `diff -w` to compare your output to the reference solutions. Credit for optional extensions is will be tracked separately and added as part of the final course grade determination.