Project 4: TAC Generation

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 P4 to be close to P3 in terms of difficulty and time consumption. We tried to give you a hand by giving you some of the pieces pre-written and littered our code with debug printing and helpful assertions. The debugging can be intense at times since you may need to drop down and 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. The back end of the compiler is where the systems-specific portion of the work comes in, so this is the project many of you have been waiting for.

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.

Starter Files

The starting files are in Piazza resources. The project contains the following files
(the boldface entries are the ones you are most likely to
modify, although depending on your strategy you may modify others as well):

`Makefile`	builds project
`main.cc`	**main** and some helper functions
`scanner.h/l`	our scanner interface/implementation
`parser.y`	**bison** parser; replace with your own
`ast.h/.cc`	interface/implementation of base AST node class
`ast_type.h/.cc`	interface/implementation of AST type classes
`ast_decl.h/.cc`	interface/implementation of AST declaration classes
`ast_expr.h/.cc`	interface/implementation of AST expression classes
`ast_stmt.h/.cc`	interface/implementation of AST statement classes
`codegen.h/.cc`	interface/implementation of **CodeGenerator** class
`tac.h/.cc`	interface/implementation of **Tac** class and subclasses
`mips.h/.cc`	interface/implementation of our provided TAC-to-MIPS translator
`errors.h/.cc`	error-reporting class for you to use
`hashtable.h/.cc`	simple hashtable template class
`list.h`	simple list template class
`location.h`	utilities for handling locations, **yylloc**/**yyltype**
`utility.h/.cc`	interface/implementation of our provided utility functions
`samples/`	directory of test input files
`run`	script to compile and execute result on SPIM simulator

Copy the entire directory to your home directory. Use **make** to build the project. It reads input from **stdin** and you can use standard UNIX file redirection to read from a file and/or save the output to a file:

```
% dcc < samples/program.decaf > program.asm
```

The output is MIPS assembly that can be executed on the SPIM simulator. The **spim** executable is the same folder . There is also an **qtspim** visual version that is in http://pages.cs.wisc.edu/~larus/spim.html. 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: ./trap.handler
hello world
```

As always, the first thing to do is to carefully read all the files we give you and make sure you understand the lay of the land. 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 you used for **Print** and **Check** in previous programming projects. You can decide how much of your P3 semantic analysis you want to incorporate into your p4 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 without the clutter.
- You are to replace **parser.y** with your own parser (or our reference parser from P3).   You should not need to make changes to the parser or rearrange the grammar, but you can if you like.
- We have removed **double**s from the types of Decaf for this project. In the generated code, we treat **bool**s 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. However, getting this working (perhaps by using multiple vtables and vtable deltas) would be an excellent way to earn some extra credit!
- 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 subclasses declared in **tac.h**. The **CodeGenerator** supports some 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

For the suggested checkpoint, you need to implement code generation for a single main function without arrays or objects. Here is a quick sketch of a reasonable 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).

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 some number of underscores 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).   Be careful not to have name collisions with our built-in routines! Also, be careful not to use labels that have the same name as a MIPS instruction; if you add prefixes to all of the function names, this should not be a problem.

- 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 particularly, 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 object's 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 debug key **tac** can be used to skip final MIPS code generation and just print the TAC instructions. The debug flag can be set with **–d tac** when invoking the program or programmatically via **SetDebugForKey**. 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 pleasure in the samples directory. 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 certainly involving making up your own additional tests.

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

Grading

The final submission is worth 15% of your overall grade in the course. We will thoroughly test your submission against many samples. We will run the object code produced your compiler on the **spim** simulator and **diff** against the correct runtime output.

Once you have finished this assignment, you'll have built a complete working compiler in a fast-paced eight weeks. Congratulations – this is no mean feat!

There are many opportunities for extra credit on this assignment, and we'd be more than happy to reward you for going above and beyond what's required. Here are a few extensions you can add into your code generator that would be really, really cool:

- **Support interfaces**.   We don't ask you to implement interfaces because of the complexities involved in adjusting object pointers. However, coding up support for interfaces will give you a much deeper understanding of how object layouts work in major programming languages.
- **Add optimization**.   Your generated code needn't be efficient for this assignment, but that doesn't mean that you can't try to optimize your generated IR. Any optimizations that you do, provided that they are correct, will earn extra credit, as long as you document it in your README.
- **Improve our code generator**.   Our code generator does a reasonable job managing registers and the stack, but perhaps you can do better! Feel free to tweak our code generator to make it more awesome. For major extra credit and lasting fame, see if you can tweak it to output x86 assembly.
- **Support switch and postfix expressions**.   In p2 we asked you to add in a **switch** statement and postfix ++ and -- operators. Why not add them to the code generator?
- **Support doubles**.   We removed the double type from this assignment to simplify the code generator logic.   Try adding them back in!

Good luck!