PP4: Code Generation

PP4 due 11:59pm 4/3/2019

Goal

In this 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 else is needed. Those TAC instructions are then translated into MIPS assembly via a translator 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.

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.

Starter files

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

`Makefile`	builds project
`main.cc`	and some helper functions
`scanner.h/l`	our scanner interface/implementation
`parser.y`	yacc parser for Decaf
`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 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

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, which actually executes the code, is in `~olivertc/Public/spim-install/`. We've also provided a shell script at `~olivertc/Public/spim`, which adds a command-line option to preload the exception handling library for SPIM, so you don't need to do it by yourself.

You can either invoke it using the full path or get a copy of the SPIM simulator from http://pages.cs.wisc.edu/~larus/spim.html.

The -file argument allows you to specify the file of MIPS assembly to execute.

```
$ ~olivertc/Public/spim -file program.asm
```

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. It will be your job to add the `Emit()` function to the nodes, most likely implemented in a polymorphic form similar to `Print()` in PP2 and `Check()` in PP3. You can decide how much of your PP3 semantic analysis you want to incorporate into your PP4 project. We will not test Decaf programs with semantic errors, so you are free to disable or remove your semantic analysis to allow you to concentrate on your new task without the clutter.
- We have removed doubles from the types of Decaf for this project. In the generated code, we treat bools just like ordinary 4-byte integers, which evaluate to 0 or not 0. The only strings are string constants, which will be referred to with pointers. Arrays and objects (variables of class type) are also implemented as pointers. Thus all variables/parameters are 4 bytes in size, which 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.
- 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.

Code generator implementation

Here is a quick sketch of a reasonable approach:

- Before you begin, go back over the TAC instructions and examples in the TAC lecture notes 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.
- Generating code for other function definitions isn't much different than it was for `main`, other than that you need to assign locations to the function parameters and figure out your strategy for assigning function labels. Our solution uses the function name prefixed with an underbar as the function label and for classes we 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. 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 your earlier object pictures 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 are to 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.

Built-In Functions

We have provided MIPS code for the following functions for you so you can generate code that calls these functions to support various functionalities in Decaf such as `New()`, `NewArray()`, `Print()`, `ReadLine()`, and string equality tests. These functions can be accessed through the `CodeGenerator::GenBuiltInCall()` function. The MIPS code for these functions are listed in `main.cc`.

`Alloc(`*n*`)`	Allocates *n* bytes in the heap and returns its address
`ReadInteger()`	Reads and returns an integer from `stdin` and returns its value
`ReadLine()`	Reads a line of at most 100 characters from `stdin` into an internal character array and returns its address
`StringEqual(`*x*`, `*y*`)`	Returns `1` if the strings point to by *x* and *y* are equal, otherwise `0`
`PrintInt(`*x*`)`	Print the integer *x* to `stdout`

`PrintString(`*x*`)`	Print the string pointed to by *x* to `stdout`
`PrintBool(`*x*`)`	Print `true` to `stdout` if *x* is not `0`, otherwise `false`
`Halt()`	Terminates the program

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.
- 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.
- We also provided the `.s` files for the sample programs, which contains sample MIPS assembly code to help you know better how the code is generated. You don't need to generate exactly the same code, but you can try to trace the assembly code when you don't know how to implement a certain feature.

Testing your compiler

There are various test files that are provided to you 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. However, the runtime output from SPIM should match exactly. Be sure to test your program thoroughly, which will almost certainly involve making up 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 90 points. We will 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.

Submission

Use the following command on any CAEN machine to submit your programs:

```
~olivertc/Public/submit.sh 4 <path_to_directory>
```

For compiling your submission, the autograder will execute `make depend`, then `make clean` followed by `make` using your `Makefile`. Note that the system is running CAEN's default version of `g++` (4.8.5).

Submissions via this method will be graded at 12:00am and 12:00pm every day, with responses being emailed back to you and your partner with the score of the submission.