# Code generation and optimization

**DEADLINE**: May 7, 23:59.
**IMPORTANT**: You should hand in your solution on the Minerva Dropbox. Create an archive using the `make_tarball.sh` script, and make sure you send it to the teaching assistants.
For any questions, contact us at `compilers@lists.ugent.be`.

## 1 Introduction

In this lab, you will implement a compiler that lowers the Cheetah AST you have worked with in the previous labs to x86-64 assembly code. Your compiler should treat the underlying system as a stack machine, i.e., it should push and pop values onto the stack and only use registers or other instructions when absolutely necessary. You should at least be able to process the source code for `fibonacci` and generate code for an executable, verifying that it computes the correct value. Other requirements are listed throughout this document.

For the second part of this assignment, you will implement a peephole optimizer that takes the generated assembly code and simplifies it. You should implement at least 3 optimizations that reduce the code size (as determined by the number of assembly instructions), and demonstrate the effectiveness of these passes on the `fibonacci` source code.

## 2 Set-up

Start by setting up the development environment for this lab (refer to previous assignments on the use of Docker), download and extract the files, and configure the project using CMake:

```
$ docker run --rm -v "$(pwd)":/files -it tbesard/compilers:pract3
$ cd /files
$ cmake .
```

If you execute `make`, the source files that make up your compiler and the supporting runtime library will be compiled. This results in two binaries: the `cheetah` executable, and a `runtime.c.o` object file. You can use the `cheetah` executable to process files written in the C subset from the previous labs and output assembly code to standard output:

```
$ ./cheetah test/dummy.c
.globl main
main:
    ...

# alternatively, outputting to a file
$ ./cheetah -o test/dummy.S test/dummy.c
```

To assemble this code, and link it together with the runtime library into an executable file, you can use the host compiler on your system:

```
$ cc CMakeFiles/runtime.dir/src/runtime.c.o -o test/dummy test/dummy.S

# verify the binary executes properly
$ ./test/dummy
```

You can automate the above steps using `make dummy`. Such a target will be created for every `*.c` file in the `test/` folder. To compile all test at once, you can use the `make test` target.

## 3 Debugging

When developing a compiler, it is easy to emit instructions that will make the resulting executable crash. In that case, you can use `gdb` to inspect the crash:

```
$ gdb ./test/dummy

# run the program
(gdb) run
Starting program: test/dummy

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554006 in ?? ()

# switch to assembly mode to inspect the faulting instruction
(gdb) layout asm

# navigate the stack frame
(gdb) up
(gdb) down
```

It can also be useful to inspect the code generated by the host compiler for C code:

```
$ gcc -S -o - something.c
        .text
        .globl  main
        .type   main, @function
main:
    ...
```

# 4 Compiler infrastructure

The files as part of this assignment already define much of the necessary functionality and infrastructure that a compiler requires. At the highest level, every AST object now has an `emit` function that takes care of emitting code for the node and any dependent nodes. You will need to complete these `emit` functions in `codegen/emit.cpp`; look for `TODO`s in the code and comments.

The `codegen.hpp` header defines three important datastructures that you will work with: `Program`, `Block` and `Instruction`. The `Program` class represents the program that is being emitted, and is accessible as the argument to each `emit` function. It contains a list of `Blocks`, which are identified by a label and contain a list of `Instructions`. These instructions have three fields: `name` for a textual representation of the instruction name, `arguments` for a list of arguments, and `comment` for an optional string that will be emitted as part of the generated code.

You can easily add instructions to the current basic block by using output operators:

```
prog << Instruction{"call", {"echo"}, "call to echo"};
```

In the case of basic blocks, you need to use a label to create the block as well as when referring to it. These labels need to be unique, and a helper function `label` is provided to facilitate that:

```
// hard-coded label
prog << Block{"my_label"};

// generated label
auto unique_label = label(prog, "my_label");
prog << Block{unique_label};
```

The assembly code generated by the `cheetah` compiler is AT&T syntax, i.e., `mov SRC DST`.

# 5 Emitting code

The compiler as it stands is only able to process the `dummy.c` source file that is part of the assignment. For the first part of this lab, you should extend the compiler until it can generate correct code for the `fibonacci` source file. You should approach this incrementally, implementing one feature at a time. To that end, the following paragraphs will explain some of these features and how you should implement them.

## 5.1 Functions calls

To be able to call functions, you need to complete the implementation of `CallExpr::emit` and implement the following features:

- emit and store arguments
- generate a call
- return a value

These features should obey the SystemV AMD64 ABI, for which there are some static values defined at the top of `codegen/emit.cpp` that should help you. By doing so, you should be able to call functions in the runtime library. This library is compiled by the host compiler, and defines two functions: `void echo(int)` and `int read()`.

Try to call the `echo` function:

```
echo(1);
```

Since we are working with a stack machine, every expression should put a value on the stack. In the case of a void function that returns nothing, you should use the `0xABCDEF` sentinel value. You can query the return type of a function in the `func_decls` map, indexed by the function name and containing a tuple of the return type and the number of argument. This map is prepopulated with entries for the `read` and `echo` function as defined in the runtime library.

### 5.2 Function declarations

The obvious next step is to be able to define your own functions:

```
void do_echo() {
    echo(1);
}
do_echo();
```

Although this code will compile already, it does not conform to the SystemV ABI. Complete the implementation of `FuncDecl::emit` and add a function prologue doing the following:

- save callee-saved registers
- set the base pointer
- align the stack pointer by 16 bytes

After function execution, the epilogue should undo these changes:

- restore the stack pointer
- restore callee-saved registers

These changes will make it possible to support local variables and function parameters.

### 5.3 Variable declarations

Now make it possible for generated code to use variables:

```
void do_echo() {
    int i = 1;
    echo(i);
}
do_echo();
```

This needs changes to the `VarDecl::emit` function:

- determine the size (you should only support `int`)
- reserve stack space, making sure the stack remains 16-byte aligned
- add the location of the variable relative to the base pointer in `var_decls`

Local variable locations are reset at the start of each function, and can be accessed using the `variable` helper function to quickly get the location of a previously-defined variable.

### 5.4 Function parameters

The next step is to use use variable declarations to add support for function parameters:

```
void do_echo(int i) {
    echo(i);
}
do_echo(1);
```

This requires another modification to `FuncDecl::emit`, emitting the variable declaration of each parameter and populating it with the argument value. To get a reference to a parameter, define a `parameter(size_t i)` function cf. the `variable` helper function from the previous paragraph. Make sure your implementation does not only work with the `fibonacci` source code, but supports any function with any number of parameters.

### 5.5 Operators

Now implement unary and binary arithmetic operators:

```
echo(-1);
echo(1 + 1);
```

For logical operations, push 1 or 0 for respectively true or false on the stack:

```
echo(1 >= 1);
```

### 5.6 Control flow

Finally, add the necessary functionality to support if tests, and both while and for loops.

## 6 Code optimization

For the second part of this assignment, you should create three peephole optimizations in `codegen/optimize.cpp` that reduce the generated code size (in number of instructions). A peephole optimization is a kind of optimization that looks for small, local patterns of instructions by inspecting the program using a sliding window. The necessary infrastructure, as well as an example optimization are already implemented in `codegen/optimize.cpp`.

The optimizer is enabled by default, and can be disabled (for comparison purposes) by invoking `cheetah` with the `-O0` flag. Make sure your optimizations are correct, and have them output relevant optimization remarks just like the example.