Introduction to Compilers and Language Design
Copyright © 2019 Douglas Thain.
Hardcover ISBN: 978-0-359-13804-3
Paperback ISBN: 978-0-359-14283-5
First edition.

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at `http://compilerbook.org`

Revision Date: February 5, 2020

Chapter 11 – Code Generation

11.1 Introduction

Congratulations, you have made it to the final stage of the compiler! After scanning and parsing the source code, constructing the AST, performing type checking, and generating an intermediate representation, we are now ready to generate some code.

To start, we are going to take a naïve approach to code generation, in which we consider each element of the program in isolation. Each expression and statement will be generated as a standalone unit, without reference to its neighbors. This is easy and straightforward, but it is conservative and will lead to a large amount of non-optimal code. But it will work, and give you a starting point for thinking about more sophisticated techniques.

The examples will focus on X86-64 assembly code, but they are not hard to adapt to ARM and other assembly languages as needed. As with previous stages, we will define a method for each element of a program. `decl_codegen` will generate code for a declaration, calling `stmt_codegen` for a statement, `expr_codegen` for an expression, and so on. These relationships are shown in Figure 11.1.

Once you have learned this basic approach to code generation, you will be ready for the *following* chapter, in which we consider more complex methods for generating more highly optimized code.

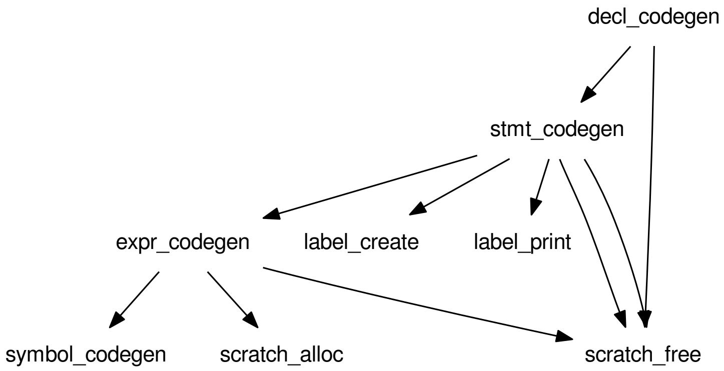
11.2 Supporting Functions

Before generating some code, we need to set up a few supporting functions to keep track of a few details. To generate expressions, you will need some **scratch registers** to hold intermediate values between operators. Write three functions with the following interface:

```
int scratch_alloc();
void scratch_free( int r );
const char * scratch_name( int r );
```

Looking back at Chapter 10, you can see that we set aside each register for a purpose: some are for function arguments, some for stack frame

Figure 11.1: Code Generation Functions



management, and some are available for scratch values. Take those scratch registers and put them into a table like this:

r	0	1	2	3	4	5	6
name	%rbx	%r10	%r11	%r12	%r13	%r14	%r15
inuse	X		X				

Then, write `scratch_alloc` to find an unused register in the table, mark it as in use, and return the register number `r`. `scratch_free` should mark the indicated register as available. `scratch_name` should return the name of a register, given its number `r`. Running out of scratch registers is possible, but unlikely, as we will see below. For now, if `scratch_alloc` cannot find a free register, just emit an error message and halt.

Next, we will need to generate a large number of unique, anonymous labels that indicate the targets of jumps and conditional branches. Write two functions to generate and display labels:

```
int label_create();
const char * label_name( int label );
```

`label_create` simply increments a global counter and returns the current value. `label_name` returns that label in a string form, so that label 15 is represented as `".L15"`.

Finally, we need a way of mapping from the symbols in a program to the assembly language code representing those symbols. For this, write a function to generate the address of a symbol:

```
const char * symbol_codegen( struct symbol *s );
```

This function returns a string which is a fragment of an instruction, representing the address computation needed for a given symbol. Write `symbol_codegen` to first examine the scope of the symbol. Global variables are easy: the name in assembly language is the same as in the source language. If you have a `symbol` structure representing the global variable `count:integer`, then `symbol_codegen` should simply return `count`.

Symbols that represent local variables and function parameters should instead return an address computation that yields the position of that local variable or parameter on the stack. The groundwork for this was laid in the typechecking phase, where you assigned each symbol a unique number, starting with the parameters and continuing with each local variable.

For example, suppose you have this function definition:

```
f: function void ( x: integer, y: integer ) =
{
    z: integer = 10;
    return x + y + z;
}
```

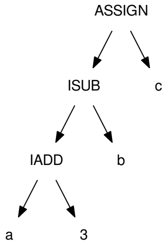
In this case, `x` was assigned a position of zero, `y` is position one, and `z` is position two. Now look back at Figure 10.5, which shows the stack layout on the X86-64 processor. Position zero is at the address `-8(%rbp)`, position one is at `-16(%rbp)`, and position two is at `-24(%rbp)`.

Given that, you can now extend `symbol_codegen` to return a string describing the precise stack address of local variables and parameters, knowing only its position in the stack frame.

11.3 Generating Expressions

The basic approach to generating assembly code for an expression is to perform a post-order traversal of the AST or DAG, and emit one or more instructions for each node. The main idea is to keep track of the registers in which each intermediate value is stored. To do this, add a `reg` field to the AST or DAG node structure, which will hold the number of a register returned by `scratch_alloc`. As you visit each node, emit an instruction and place into the `reg` field the number of the register containing that value. When the node is no longer needed, call `scratch_free` to release it.

Suppose we want to generate X86 code for the following DAG, where a, b and c are global integers:



```
MOVQ  a, R0
MOVQ $3, R1
ADDQ R0, R1
MOVQ  b, R0
SUBQ R0, R1
MOVQ R1, c
```

Figure 11.2: Example of Generating X86 Code from a DAG

A post-order traversal would visit the nodes in the following order:

1. Visit the a node. Call scratch_alloc to allocate a new register (0) and save that in node->reg. Then emit the instruction MOVQ a, R0 to load the value into register zero. [1]

2. Visit the 3 node. Call scratch_alloc to allocate a new register (1) and emit the instruction MOVQ $3, R1.

3. Visit the IADD node. By examining the two children of this node, we can see that their values are stored in registers R0 and R1, so we emit an instruction that adds them together: ADDQ R0, R1. This is a destructive two-address instruction which leaves the result in R1. R0 is no longer needed, so we call scratch_free(0).

4. Visit the b node. Call scratch_alloc to allocate a new register (0) and emit MOVQ b, R0.

5. Visit the ISUB node. Emit SUBQ R0, R1, leaving the result in R1, and freeing register R0.

6. Visit the c node, but don't emit anything, because it is the target of the assignment.

7. Visit the ASSIGN node and emit MOVQ R1, c.

[1] Note that the actual name of register R0 is scratch_name(0), which is %rbx. To keep the example clear, we will call them R0, R1, etc. for now.

And here is the same code with the actual register names provided by `scratch_name`:

```
MOVQ    a, %rbx
MOVQ   $3, %r10
ADDQ %r10, %rbx
MOVQ    b, %rbx
SUBQ %rbx, %r10
MOVQ %r10, c
```

Here is how to implement it in code. Write a function called `expr_codegen` that first recursively calls `expr_codegen` for its left and right children. This will cause each child to generate code such that the result will be left in the register number noted in the `register` field. The current node then generates code using those registers, and frees the registers it no longer needs. Figure 11.3 gives a skeleton for this approach.

A few additional refinements are needed to the basic process.

First, not all symbols are simple global variables. When a symbol forms part of an instruction, use `symbol_codegen` to return the string that gives the specific address for that symbol. For example, if `a` was the first parameter to the function, then the first instruction in the sequence would have looked like this instead:

```
MOVQ -8(%rbp), %rbx
```

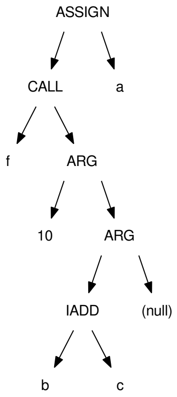
Second, some nodes in the DAG may require multiple instructions, so as to handle peculiarities of the instruction set. You will recall that the X86 `IMUL` only takes one argument, because the first argument is always `%rax` and the result is always placed in `%rax` with the overflow in `%rdx`. To perform the multiply, we must move one child register into `%rax`, multiply by the other child register, and then move the result from `%rax` to the destination scratch register. For example, the expression `(x*10)` would translate as this:

```
MOV  $10, %rbx
MOV  x,   %r10
MOV  %rbx, %rax
IMUL %r10
MOV  %rax, %r11
```

Of course, this also means that `%rax` and `%rdx` cannot be used for other purposes while a multiply is in progress. Since we have a large number of scratch registers to work with, we will just not use `%rdx` for any other purpose in our basic code generator.

```
void expr_codegen( struct expr *e )
{
    if(!e) return;

    switch(e->kind) {

        // Leaf node: allocate register and load value.

        case EXPR_NAME:
            e->reg = scratch_alloc();
            printf("MOVQ %s, %s\n",
                symbol_codegen(e->symbol),
                scratch_name(e->reg));
            break;

        // Interior node: generate children, then add them.

        case EXPR_ADD:
            expr_codegen(e->left);
            expr_codegen(e->right);
            printf("ADDQ %s, %s\n",
                scratch_name(e->left->reg),
                scratch_name(e->right->reg));
            e->reg = e->right->reg;
            scratch_free(e->left->reg);
            break;

        case EXPR_ASSIGN:
            expr_codegen(e->left);
            printf("MOVQ %s, %s\n",
                scratch_name(e->left->reg),
                symbol_codegen(e->right->symbol));
            e->reg = e->left->reg;
            break;
        . . .
    }
}
```

Figure 11.3: Expression Generation Skeleton



```
MOVQ  $10, %rbx
MOVQ  b, %r10
MOVQ  c, %r11
ADDQ  %r10, %r11
MOVQ  %r11, %rsi
MOVQ  %rbx, %rdi
PUSHQ %r10
PUSHQ %r11
CALL  f
POPQ  %r11
POPQ  %r10
MOVQ  %rax, %rbx
MOVQ  %rbx, a
```

Figure 11.4: Generating Code for a Function Call

Third, how do we invoke a function? Recall that a function is represented by a single CALL node, with each of the arguments in an unbalanced tree of ARG nodes. Figure 11.4 gives an example of a DAG representing the expression `a=f(10,b+c)`.

The code generator must evaluate each of the ARG nodes, computing the value of each left hand child. If the machine has a stack calling convention, then each ARG node corresponds to a PUSH onto the stack. If the machine has a register calling convention, generate all of the arguments, then copy each one into the appropriate argument register after all arguments are generated. Then, emit CALL on the function name, after saving any caller-saved registers. When the function call returns, move the value from the return register (`%rax`) into a newly allocated scratch register and restore the caller-saved registers.

Finally, note carefully the side effects of expressions. Every expression has a **value** which is computed and left in a scratch register for its parent node to consider. Some expressions also have **side effects** which are actions in addition to the value. With some operators, it is easy to overlook one or the other!

For example, the expression `(x=10)` yields a *value* of 10, which means you can use that expression anywhere a value is expected. This is what allows you to write `y=x=10` or `f(x=10)`. The expression also has the *side effect* of storing the value 10 into the variable x. When you generate code for `x=10` assignment, be sure to carry out the side effect of storing 10 into x (that's obvious) but also retain the value 10 in the register that represents the value of the expression.

11.4 Generating Statements

Now that we have encapsulated expression generation into a single function expr_codegen, we can begin to build up larger code structures that rely upon expressions. stmt_codegen will create code for all control flow statements. Begin by writing a skeleton for stmt_codegen like this:

```
void stmt_codegen( struct stmt *s )
{
    if(!s) return;

    switch(s->kind) {
        case STMT_EXPR:
            ...
            break;
        case STMT_DECL:
            ...
            break;
        ...
    }
    stmt_codegen(s->next);
}
```

Figure 11.5: Statement Generator Skeleton

Now consider each kind of statement in turn, starting with the easy cases. If the statement describes a declaration STMT_DECL of a local variable, then just delegate that by calling decl_codegen:

```
case STMT_DECL:
    decl_codegen(s->decl);
    break;
```

A statement that consists of an expression (STMT_EXPR) simply requires that we call expr_codegen on that expression, and then free the scratch register containing the top-level value of the expression. (In fact, every time expr_codegen is called, a scratch register should be freed.)

```
case STMT_EXPR:
    expr_codegen(s->expr);
    scratch_free(s->expr->reg);
    break;
```

A return statement must evaluate an expression, move it into the designated register for return values %rax, and then jump to the function epilogue, which will unwind the stack and return to the call point. (See below for more details about the prologue.)

```
case STMT_RETURN:
    expr_codegen(s->expr);
    printf("MOV %s, %%rax\n",scratch_name(s->expr->reg));
    printf("JMP .%s_epilogue\n",function_name);
    scratch_free(s->expr->reg);
    break;
```

(The careful reader will notice that this code needs to know the name of the function that contains this statement. You will have to figure out a way to pass that information down.)

Control flow statements are more interesting. It's useful to first consider what we want the output assembly language to look like, and then work backwards to get the code for generating it.

Here is a template for a conditional statement:

if ($\boxed{\textit{expr}}$ **)** {
    $\boxed{\textit{true-statements}}$
} **else** {
    $\boxed{\textit{false-statements}}$
}

To express this in assembly, we must evaluate the control expression so that its value resides in a known register. A `CMP` expression is then used to test if the value is equal to zero (false). If the expression is false, then we must jump to the false branch of the statement with a `JE` (jump-if-equal) statement. Otherwise, we continue through to the true branch of the statement. At the end of the true statement, we must `JMP` over the else body to the end of the statement.

    $\boxed{\textit{expr}}$
    `CMP $0`, *register*
    `JE` *false-label*
    $\boxed{\textit{true-statements}}$
    `JMP` *done-label*
false-label **:**
    $\boxed{\textit{false-statements}}$
done-label **:**

Once you have a skeleton of the desired code, writing the code generator is easy. First, generate two new labels, then call `expr_codegen` for each expression, `stmt_codegen` for each statement, and substitute the few additional instructions as needed to make the overall structure.

```
case STMT_IF:
    int else_label = label_create();
    int done_label = label_create();
    expr_codegen(s->expr);
    printf("CMP $0, %s\n",scratch_name(s->expr->reg));
    scratch_free(s->expr->reg);
    printf("JE  %s\n",label_name(else_label));
    stmt_codegen(s->body);
    printf("JMP %s\n",label_name(done_label));
    printf("%s:\n",label_name(else_label));
    stmt_codegen(s->else_body);
    printf("%s:\n",label_name(done_label));
    break;
```

A similar approach can be used to generate loops. Here is the source template of a for-loop:

for ($\boxed{\textit{init-expr}}$ **;** $\boxed{\textit{expr}}$ **;** $\boxed{\textit{next-expr}}$ **)** {
    $\boxed{\textit{body-statements}}$
}

And here is the corresponding assembly template. First, evaluate the initializing expression. Then, upon each iteration of the loop, evaluate the control expression. If false, jump to the end of the loop. If not, execute the body of the loop, then evaluate the next expression.

    $\boxed{\textit{init-expr}}$
top-label:
    $\boxed{\textit{expr}}$
    `CMP $0,` *register*
    `JE` *done-label*
    $\boxed{\textit{body-statements}}$
    $\boxed{\textit{next-expression}}$
    `JMP` *top-label*
done-label :

Writing the code generator is left as an exercise for the reader. Just keep in mind that each of the three expressions in a for-loop can be omitted. If the *init-expr* or the *next-expr* are omitted, they have no effect. If the *expr* is omitted, it is assumed to be true. [2]

Many languages have loop-control constructs like `continue;` and `break;`. In these cases, the compiler must keep track of the labels associated with the current loop being generated, and convert these into a `JMP` to the top label, or the done-label respectively.

The `print` statement in B-Minor is a special case of an imperative statement with variant behavior depending on the type of the expression to be printed. For example, the following `print` statement must generate slightly different code for the integer, boolean, and string to be printed:

```
i: integer = 10;
b: boolean = true;
s: string = "\n";
print i, b, s;
```

Obviously, there is no simple assembly code corresponding to the display of an integer. In this case, a common approach is to reduce the task into an abstraction that we already know. The printing of integers, booleans, strings, etc., can be delegated to function calls that explicitly perform those actions. The generated code for `print i, b, s` is then equivalent to this:

```
print_integer(i);
print_boolean(b);
print_string(s);
```

So, to generate a `print` statement, we simply generate the code for each expression to be printed, determine the type of the expression with `expr_typecheck` and then emit the corresponding function call.

Of course, each of these functions must be written and then linked into each instance of a B-Minor program, so they are available when needed. These functions, and any others necessary, are collectively known as the **runtime library** for B-Minor programs. As a general rule, the more high-level a programming language, the more runtime support is needed.

[2] Yes, each of the three components of a for-loop are expressions. It is customary that the first has a side effect of initialization (`i=0`), the second is a comparison (`i<10`), and the third has a side effect to generate the next value (`i++`), but they are all just plain expressions.

11.5 Conditional Expressions

Now that you know how to generate control flow statements, we must return to one aspect of expression generation. Conditional expressions (less-than, greater-than, equals, etc.) compare two values and return a boolean value. They most frequently appear in control flow expressions but can also be used as simple values, like this:

```
b: boolean = x < y;
```

The problem is that there is no single instruction that simply performs the comparison and places the boolean result in a register. Instead, you must go the long way around and make a control flow construct that compares the two expressions, then constructs the desired result.

For example, if you have a conditional expression like this:

$\boxed{\textit{left-expr}} < \boxed{\textit{right-expr}}$

then generate code according to this template:

```
    left-expr
    right-expr
    CMP left-register, right-register
    JLT true-label
    MOV false, result-register
    JMP done-label
true-label:
    MOV true, result-register
done-label:
```

Of course, for different conditional operators, use a different jump instruction in the appropriate place. With slight variations, you can use the same approach to implement the ternary conditional operator `(x?a:b)` found in many languages.

A funny outcome of this approach is that if you generate code for an if-statement like `if(x>y){...}` in the obvious way, you will end up with *two* conditional structures in the assembly code. The first conditional computes the result of `x>y` and places that in a register. The second conditional compares that results against zero and then jumps to the true or false branch of the statement. With a little careful coding, you can check for this common case and generate a single conditional statement that evaluates the expression and uses one conditional jump to implement the statement.

11.6 Generating Declarations

Finally, emitting the entire program is a matter of traversing each declaration of code or data and emitting its basic structure. Declarations can fall into three cases: global variable declarations, local variable declarations, and global function declarations. (B-Minor does not permit local function declarations.)

Global data declarations are simply a matter of emitting a label along with a suitable directive that reserves the necessary space, and an initializer, if needed. For example, these B-Minor declarations at global scope:

```
i: integer = 10;
s: string = "hello";
b: array [4] boolean = {true, false, true, false};
```

Should yield these output directives:

```
.data
i:  .quad 10
s:  .string "hello"
b:  .quad 1, 0, 1, 0
```

Note that a global variable declaration can only be initialized by a constant value (and not a general expression) precisely because the data section of the program can only contain constants (and not code). If the programmer accidentally put code into the initializer, then the typechecker should have discovered that and raised an error before code generation began.

Emitting a local variable declaration is much easier. (This only happens when `decl_codegen` is called by `stmt_codegen` inside of a function declaration.) Here, you can assume that space for the local variable has already been established by the function prologue, so no stack manipulations are necessary. However, if the variable declaration has an initializing expression (`x:integer=y*10;`) then you must generate code for the expression, store it in the local variable, and free the register.

Function declarations are the final piece. To generate a function, you must emit a label with the function's name, followed by the function prologue. The prologue must take into account the number of parameters and local variables, making the appropriate amount of space on the stack. Next comes the body of the function, followed by the function epilogue. The epilogue should have a unique label so that `return` statements can easily jump there.

11.7 Exercises

1. Write a legal expression that would exhaust the six available scratch registers, if using the technique described in this chapter. In general, how many registers are needed to generate code for an arbitrary expression tree?

2. When using a register calling convention, why is it necessary to generate values for *all* the function arguments before moving the values into the argument registers?

3. Can a global variable declaration have a non-constant initializing expression? Explain why.

4. Suppose B-Minor included a `switch` statement. Sketch out two different assembly language templates for implementing a `switch`.

5. Write the complete code generator for the X86-64 architecture, as outlined in this chapter.

6. Write several test programs to test all the aspects of B-Minor then use your compiler to build, test, and run them.

7. Compare the assembly output of your compiler on your test programs to the output of a production compiler like `gcc` on equivalent programs written in C. What differences do you see?

8. Add an extra code generator to your compiler that emits a different assembly language like ARM or an intermediate representation like LLVM. Describe any changes in approach that were necessary.