COEN 175

Lecture 20: Code Generation Improvements

Example

- Generate code for the body of the function:

```
int b;

int f(int a)
{
    return a + b * 3;
}
```

```
movl   b, %eax
imull  $3, %eax
movl   8(%ebp), %ecx
addl   %eax, %ecx
movl   %ecx, %eax
jmp    f.exit
```

- After we load `EAX` with the return value, we need to jump to the epilogue of the function.
  - For simplicity, we will just label the epilogue of a function named `f` as `f.exit`.
- Let's look at the expression `a + b * 3` in detail.

Example: Step-By-Step

Code Executed	Node Changes	Output Produced
`Add::generate()`		
`  Identifier::generate()`	`a::_operand = "8(%ebp)"`	
`  Multiply::generate()`		
`    Identifier::generate()`	`b::_operand = "b"`	
`    Integer::generate()`	`3::_operand = "$3"`	
`    load(left,eax)`	`b::_register = eax`	`movl  b, %eax`
`    cout << … << endl`		`imull $3, %eax`
`    assign(right,nullptr)`		
`    assign(this,eax)`	`b::_register = nullptr` `*::_register = eax`	
`  load(left,ecx)`	`a::_register = ecx`	`movl  8(%ebp), %ecx`
`  cout << … << endl`		`addl  %eax, %ecx`
`  assign(right,nullptr)`	`*::_register = nullptr`	
`  assign(this,ecx)`	`a::_register = nullptr` `+::_register = ecx`	

Testing Expressions

- Sometimes the result of an expression is used solely as a truth value.
- This situation frequently arises with the comparison and logical operators.
  - For example, in Java, these operators return a `bool` rather than an `int`, and all conditional tests require a `bool`.
- Currently, we do a lot of work to compute a 0 or 1 as the result of a comparison or logical operator.
- If we are merely going to test the truth value of the result next, we can streamline this process.

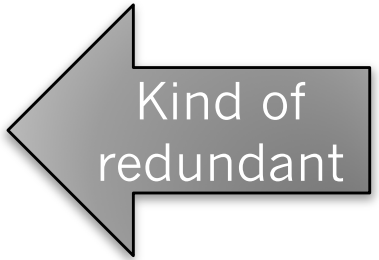
Example

- Generate code for the body of the function:

```
int a, b;

int f(void)
{
    while (a < b)
        a = a + b;
}
```

```
.L0:
        movl    a, %eax
        cmpl    b, %eax
        setl    %al
        movzbl  %al, %eax
        cmpl    $0, %eax
        je      .L1
        movl    a, %eax
        addl    b, %eax
        movl    %eax, a
        jmp     .L0
.L1:
```



Label: Header File

- Let's introduce a `Label` class to make things easier.

```
class Label {
    static unsigned _counter;
    unsigned _number;

public:
    Label();
    unsigned number() const;
};

ostream &operator <<(ostream &ostr, const Label &label);
```

Label: Source File

- Let's introduce a `Label` class to make things easier.

```
unsigned Label::_counter = 0;

Label::Label() {
    _number = _counter ++;
}

unsigned Label::number() {
    return _number;
}

ostream &operator <<(ostream &ostr, const Label &label) {
    return ostr << ".L" << label.number();
}
```

Label: Description

- Our `Label` class is essentially a wrapper around a single integer, which represents the label number.
- However, since it is a new type, we can define a constructor for it and also overload operators.
- Our constructor simply assigns it the next label number in sequence.
- The stream operator simply outputs the label number prefixed by the standard label prefix.
  - This is useful if we decide to use a different prefix.

A New Function

- Now, let's add a function `test` to `Expression`.

```
void Expression::test(const Label &label, bool ifTrue)
{
    generate();

    if (_register == nullptr)
        load(this, getreg());

    cout << "\tcmpl\t$0, " << this << endl;
    cout << (ifTrue ? "\tjne\t" : "\tje\t") << label << endl;

    assign(this, nullptr);
}
```

Explanation of Our Function

- Our test function does the following:
  1. Generates code for the expression.
  2. Compares the result against zero.
  3. Branches to the given `label` depending on the status of the `ifTrue` parameter.
- Other code generation functions can use our new function to test the truth value of an expression.
- Since our function is defined in the base class, it will be inherited by all subclasses of `Expression`.
  - Subclasses can provide their own versions, as we shall see.

Using the New Infrastructure

- We can easily write a function to generate code for a **while** statement using our new infrastructure.

```
void While::generate()
{
    Label loop, exit;

    cout << loop << ":" << endl;

    _expr->test(exit, false);
    _stmt->generate();
    release();

    cout << "\tjmp\t" << loop << endl;
    cout << exit << ":" << endl;
}
```

Releasing Registers

- We will write a convenience function to release all registers just in case we forgot somewhere.
- We can use this new function after we generate code for a statement.

```
void release()
{
    for (unsigned i = 0; i < registers.size(); i ++)
        assign(nullptr, registers[i]);
}
```

Specialized Functions

- We can write specialized versions of `test` for any subclass of `Expression`.

```
void LessThan::test(const Label &label, bool ifTrue)
{
    _left->generate();
    _right->generate();

    if (_left->_register == nullptr)
        load(_left, getreg());

    cout << "\tcmpl\t" << _right << ", " << _left << endl;
    cout << (ifTrue ? "\tjl\t" : "\tjge\t") << label << endl;

    assign(_left, nullptr);
    assign(_right, nullptr);
}
```

Example: Improved

- Generate code for the body of the function:

```
int a, b;

int f(void)
{
    while (a < b)
        a = a + b;
}
```

```
.L0:
        movl    a, %eax
        cmpl    b, %eax
        jge     .L1
        movl    a, %eax
        addl    b, %eax
        movl    %eax, a
        jmp     .L0
.L1:
```

Optional Improvements

- We could select a different register to spill than the first one in `getreg()`.
  - Which register to choose? Random? LRU?
- Instead of always spilling in `load()`, we could try to move the value to another register.
  - This gets tricky ... there's no point in moving a value in a caller-saved register to another one when doing a call.
- We can use the callee-saved registers.
  - GCC will do this to save caller-saved registers before making a function call. Clang just spills the caller-saved registers.