Lecture 19

Code Generation for Procedures in WLM

CS 241: Foundations of Sequential Programs
Winter 2018

Troy Vasiga et al
University of Waterloo

Recap

What we know to this point:

- semantic/context-sensitive analysis phase:
  - symbol table (to keep track of variable declarations)
  - ensure that variables declared exactly once
- code generation phase:
  - each grammar rule will represent one function/procedure/step in our compilation process
  - the parse tree indicates which subtrees need to produced the code
  - add comments to your generated (MIPS) code
  - it is easy to destroy register values if you are not careful

A10P5: recursively calling `wain`

- we must have multiple frames
- notice that if we call a procedure, we must use `jalr` so save $31
- treat `wain` like any other procedure! (Just like we treated $1 and $2 just like any other variable!)

A picture of the stack after `wain` calls `wain`

Suppose we have an expression like:

```
x = 3 * wain(a-1, b-1);
```

The prologue for `wain`

The epilogue for `wain`

A10P6: procedures, in general

Solving "duplicate/undeclared" procedures

Obtaining and storing signatures

What does each procedure need to do?

Each procedure has 0, 1 or 2 parameters. We can assume that parameters are in $1 and $2.

Each procedure needs

- to set its own frame pointer ($29)
- to return to its caller (i.e., `jr $31`)

What's my name again?

If names of procedures map to labels, then...

Big picture: what the MIPS output will look like

More advanced context-sensitive analysis and code generation

"Real" languages add more complexity.

- types (including casting)
- objects (including inheritence and polymorphism)
- pointers
- arrays
- arbtitrary number of parameters
- structures, templates, ...
- More sophisticated assembly languages (x86, ARM, ...)

Bonus-type things

Optimization

Large and interesting topics: we will just skim.
Recall that we can have an infinite number of MIPS programs that are equivalent to a particular WLM program. What are the "good" MIPS programs that we would like to generate?

- Increasing speed of instructions in actual code: hard

- Reducing number of statements (stack usages, register allocation)

Optimization: Constant Folding

Consider an expression 5+3.

What our compiler currently does:

What it could do:

Optimization: Constant Propagation

```
int x = 1;
return x+x;
```

What our compiler currently does:

Or, we could use the fact that we know `x = 1` always and:

Further, if `x` is never used anywhere else:

Optimization: Common subexpression elimination

Imagine you had:

```
(a+b) * (a+b)
```

Optimization: Dead-code elimination

What makes code dead?

Optimization: Register allocation

- Notice that many registers (i.e., $14 through $28) are not used by our compilation system.
- Notice that using registers would save instructions. Why?

- Suppose we could store 15 variables. Which ones would we store?

- & do you see a problem?

Optimization: Strength Reduction

- `add` is usually faster than `mult` (in the real world)
- however, in CS241, if we just count MIPS instructions, consider multiplying by 2

Optimization: Inlining Procedures

```
int f(int x) { return x+x; }
int wain(int a, int b) { return f(a); }
```

Pros/Cons:

Optimization: Tail Recursion in Procedures

In WLM, we must have exactly one `return` as the last statement in every function.

In other programming languages, you can say something like:

```
int fact(int n, int a) {
  if (n==0) return a;
  else return fact(n-1, n*a);
}
```

Notice that the last thing the function does is returning a value: there is no additional work to be done. Thus, the frame can be **reused**!

Optimization: Peephole Optimization

We may frequently run into generated MIPS like:

```
<load something simple into $3>
push $3
<load something simple into $3 which doesn't change $5>
pop $5
<combine $3 and $5>
```

A few words about overloading

```
int f(int a) { ...  }
int f(int a, int* b) { ...  }
```

Solved by *name mangling*:

Problems: C++

Solutions: `extern`