CS 516: COMPILERS

Lecture 5

Topics

- Directly compiling SIMPLE to X86.

Materials

- lec05.zip (compile1, compile2)

Announcements

- Homework 2: X86 Simulator
  - Due: Thursday, February 9th at 11:59:59pm
  - Pair-programming
- Quiz next week
  - Should last about 15 minutes
- Today Part 1:
  - Intermediate representations
- Today Part 2:
  - Simple Let IR "IR1" in `ir1.ml`
  - Translate expressions to IR1 by hand – see `ir-by-hand.ml`
  - Compiling to IR1 – see compile in `ir1.ml`
  - IR2 and beyond …

previously on...

CS 516

COMPILERS

Directly Translating AST to Assembly

- For simple languages, no need for intermediate representation.
  - e.g. the arithmetic expression language from

- Main Idea: Maintain invariants
  - e.g. Code emitted for a given expression computes the answer into rax

- Key Challenges:
  - storing intermediate values needed to compute complex expressions
  - some instructions use specific registers (e.g. shift)

see compile1 in compile.ml in lec05.zip

DIRECTLY GENERATING X86

One Simple Strategy

- Compilation is ***the process of "emitting" instructions into an instruction stream***.
- To compile an expression, we recursively compile sub expressions and then process the results.

- Invariants:
  - Compilation of an expression yields its result in rax
  - Argument (Xi) is stored in a dedicated operand
  - Intermediate values are pushed onto the stack
  - Stack slot is popped after use (so the space is reclaimed)

- Resulting code is wrapped to comply with cdecl calling conventions.

- See the compile.ml compile1.

code demo

COMPILE1

1. A first attempt

```
unzip lec05.zip ; cd lec05/ ; make
```

A. Look at the Makefile

B. **`compile1`** in `compile.ml` generates X86

C. gcc to compile & link with runtime.c

D. Run the resulting linked exectuable:
`./calculator 1 2 3 4 5 6 7 8`

compile1 on src1

```
(* Example source program expressions ------------          ------- *)

(* X1 + (X2 + (X3 + (X4  (X5 + (X6 + (X6 + (X7 +
let src1 : Compile.exp =
  (Add(Var "X1",
     Add(Var "X2",
        Add(Var "X3",
           Add(Var "X4",
              Add(Var "X5",
                 Add(Var "X6",
                    Add(Var "X7",
                       Add(Var "X8", Cons
```

Prologue

First Var exp

Save it

RHS of outermost Add

Move result of RHS from rax to r10

Calculate outermost Add and save to rax

```
    .text
    .globl  program
program:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, %rax
    pushq   %rax
    movq    %rsi, %rax
    pushq   %rax
    movq    %rdx, %rax
    pushq   %rax
    movq    %rcx, %rax
    pushq   %rax
    movq    %r8 , %rax
    pushq   %rax
    movq    %r9 , %rax
    pushq   %rax
    movq    16(%rbp), %rax
    pushq   %rax
    movq    24(%rbp), %rax
    pushq   %rax
    movq    $341, %rax
    popq    %r10
    addq    %r10, %rax
    popq    %r10
    addq    %r10, %rax
```

compile1 on src1

- This was a **First Attempt.** compile1 invariant:
  - Code emitted for a given expression computes the answer into rax

- Handy having things in RAX

- Downside: you fetch things before you need them
  - Loaded var X1 first, but needed it last.

- **Second attempt**: Rather than having the result computed into RAX, how about using the Stack?

STACK-BASED IR

compile2

- Stack-based IR.
- **Phase 1**: Convert the AST to a “flat” representation:
  - Instead of (Add(Mul(Var(x),Var(y)),42))…
  - IVar(x) ; IVar(y) ; IMul ; 42 ; IAdd // IR “instructions”
- **Phase 2:** Compile this **sequence** into X86 instructions

- To do this we will:
  - Make some handy IR data structures (insn)
  - Emit instructions that use the stack for computation
    - Consume intermediate values from the stack
    - Perform operation
    - Push new values onto the stack

Compile2 on src1

```
(* Example source program expressions ---------                   ------ *)

(* X1 + (X2 + (X3 + (X4  (X5 + (X6 + (X6 + (X7 +
let src1 : Compile.exp =
  (Add(Var "X1",
      Add(Var "X2",
          Add(Var "X3",
              Add(Var "X4",
                  Add(Var "X5",
                      Add(Var "X6",
                          Add(Var "X7",
                              Add(Var "X8"
```

Prologue

First Var exp, will be used last

Similarly the other Vars

IAdd for innermost

IAdd for second innermost

```
    .text
    .globl  program
program:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rdi
    pushq   %rsi
    pushq   %rdx
    pushq   %rcx
    pushq   %r8
    pushq   %r9
    pushq   16(%rbp)
    pushq   24(%rbp)
    pushq   $341
    popq    %rax
    popq    %r10
    addq    %r10, %rax
    pushq   %rax
    popq    %rax
    popq    %r10
    addq    %r10, %rax
    pushq   %rax
    popq    %rax
    popq    %r10
    addq    %r10, %rax
    pushq   %rax
```

Compile2 on src1

```
(* Example source program expressions ---------- ------ *)

(* X1 + (X2 + (X3 + (X4  (X5 + (X6 + (X6 + (X7 +
let src1 : Compile.exp =
  (Add(Var "X1",
     Add(Var "X2",
       Add(Var "X3",
         Add(Var "X4",
           Add(Var "X5",
             Add(Var "X6",
               Add(Var "X7",
                 Add(Var "X8"
```

Prologue

First Var exp, will be used last

Similarly the other Vars

IAdd for innermost

IAdd for second innermost

This was silly

```
    .text
    .globl  program
program:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rdi
    pushq   %rsi
    pushq   %rdx
    pushq   %rcx
    pushq   %r8
    pushq   %r9
    pushq   16(%rbp)
    pushq   24(%rbp)
    pushq   $341
    popq    %rax
    popq    %r10
    addq    %r10, %rax
    pushq   %rax
    popq    %rax
    popq    %r10
    addq    %r10, %rax
    pushq   %rax
    popq    %rax
    popq    %r10
    addq    %r10, %rax
    pushq   %rax
```

Compile2 on src1

```
(* Example source program expressions ----------                    ------ *)

(* X1 + (X2 + (X3 + (X4  (X5 + (X6 + (X6 + (X7 +
let src1 : Compile.exp =
  (Add(Var "X1",
       Add(Var "X2",
           Add(Var "X3",
               Add(Var "X4",
                   Add(Var "X5",
                       Add(Var "X6",
                           Add(Var "X7",
                               Add(Var "X8"
```

Prologue

First Var exp, will be used last

Similarly the other Vars

IAdd for innermost

IAdd for second innermost

```
    .text
    .globl  program
program:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rdi
    pushq   %rsi
    pushq   %rdx
    pushq   %rcx
    pushq   %r8
    pushq   %r9
    pushq   16(%rbp)
    pushq   24(%rbp)
    pushq   $341
    popq    %rax
    popq    %r10
    addq    %r10, %rax
    pushq   %rax
    popq    %rax
    popq    %r10
    addq    %r10, %rax
    pushq   %rax
    popq    %rax
    popq    %r10
    addq    %r10, %rax
    pushq   %rax
```

Lots of memory usage for stuff that was in registers

This was silly

INTERMEDIATE REPRESENTATIONS

compile1 on src1

```
(* Example source program expressions ----------                ------ *)

(* X1 + (X2 + (X3 + (X4  (X5 + (X6 + (X6 + (X7 +
let src1 : Compile.exp =
  (Add(Var "X1",
     Add(Var "X2",
        Add(Var "X3",
           Add(Var "X4",
              Add(Var "X5",
                 Add(Var "X6",
                    Add(Var "X7",
                       Add(Var "X8", Cons
```

Prologue

First Var exp

Save it

RHS of outermost Add

Move result of RHS from rax to r10

Calculate outermost Add and save to rax

```
      .text
      .globl   program
program:
      pushq    %rbp
      movq     %rsp, %rbp
      movq     %rdi, %rax
      pushq    %rax
      movq     %rsi, %rax
      pushq    %rax
      movq     %rdx, %rax
      pushq    %rax
      movq     %rcx, %rax
      pushq    %rax
      movq     %r8 , %rax
      pushq    %rax
      movq     %r9 , %rax
      pushq    %rax
      movq     16(%rbp), %rax
      pushq    %rax
      movq     24(%rbp), %rax
      pushq    %rax
      movq     $341, %rax
      popq     %r10
      addq     %r10, %rax
      popq     %r10
      addq     %r10, %rax
```

Why do something else?

Why do something else?

- This is a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

Why do something else?

- This is a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

Why do something else?

- This is a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

But…

- The resulting code quality is poor.

Why do something else?

- This is a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

But…

- The resulting code quality is poor.
- Richer source language features are hard to encode
  - Structured data types, objects, first-class functions, etc.

Why do something else?

- This is a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

But…

- The resulting code quality is poor.
- Richer source language features are hard to encode
  - Structured data types, objects, first-class functions, etc.
- It's hard to optimize the resulting assembly code.
  - The representation is too concrete – e.g. it has committed to using certain registers and the stack
  - Only a fixed number of registers
  - Some instructions have restrictions on where the operands are located

Why do something else?

- This is a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

But…

- The resulting code quality is poor.
- Richer source language features are hard to encode
  - Structured data types, objects, first-class functions, etc.
- It's hard to optimize the resulting assembly code.
  - The representation is too concrete – e.g. it has committed to using certain registers and the stack
  - Only a fixed number of registers
  - Some instructions have restrictions on where the operands are located
- Control-flow is not structured:
  - Arbitrary jumps from one code block to another
  - Implicit fall-through makes sequences of code non-modular (i.e. you can't rearrange sequences of code easily)

Why do something else?

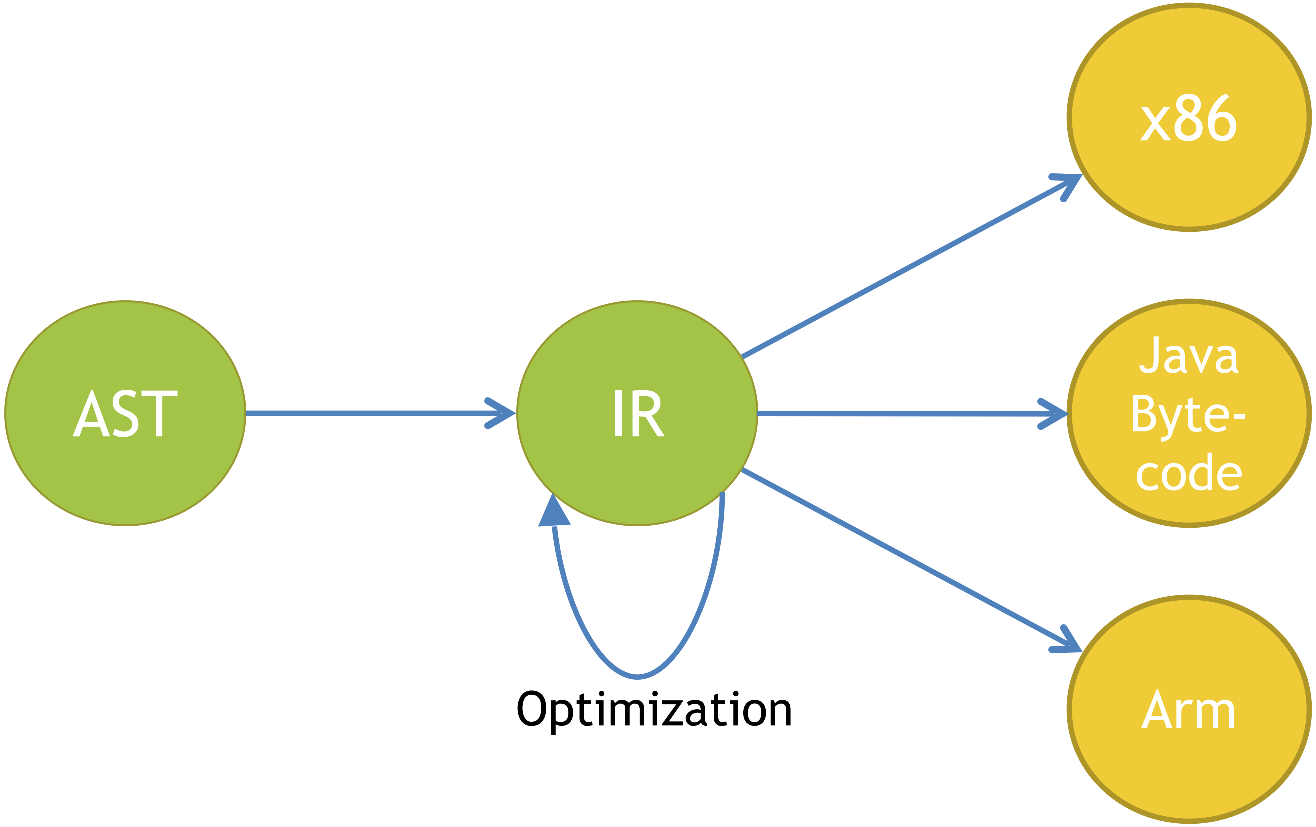
- This is a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

But…

- The resulting code quality is poor.
- Richer source language features are hard to encode
  - Structured data types, objects, first-class functions, etc.
- It's hard to optimize the resulting assembly code.
  - The representation is too concrete – e.g. it has committed to using certain registers and the stack
  - Only a fixed number of registers
  - Some instructions have restrictions on where the operands are located
- Control-flow is not structured:
  - Arbitrary jumps from one code block to another
  - Implicit fall-through makes sequences of code non-modular (i.e. you can't rearrange sequences of code easily)
- Retargeting the compiler to a new architecture is hard.
  - Target assembly code is hard-wired into the translation

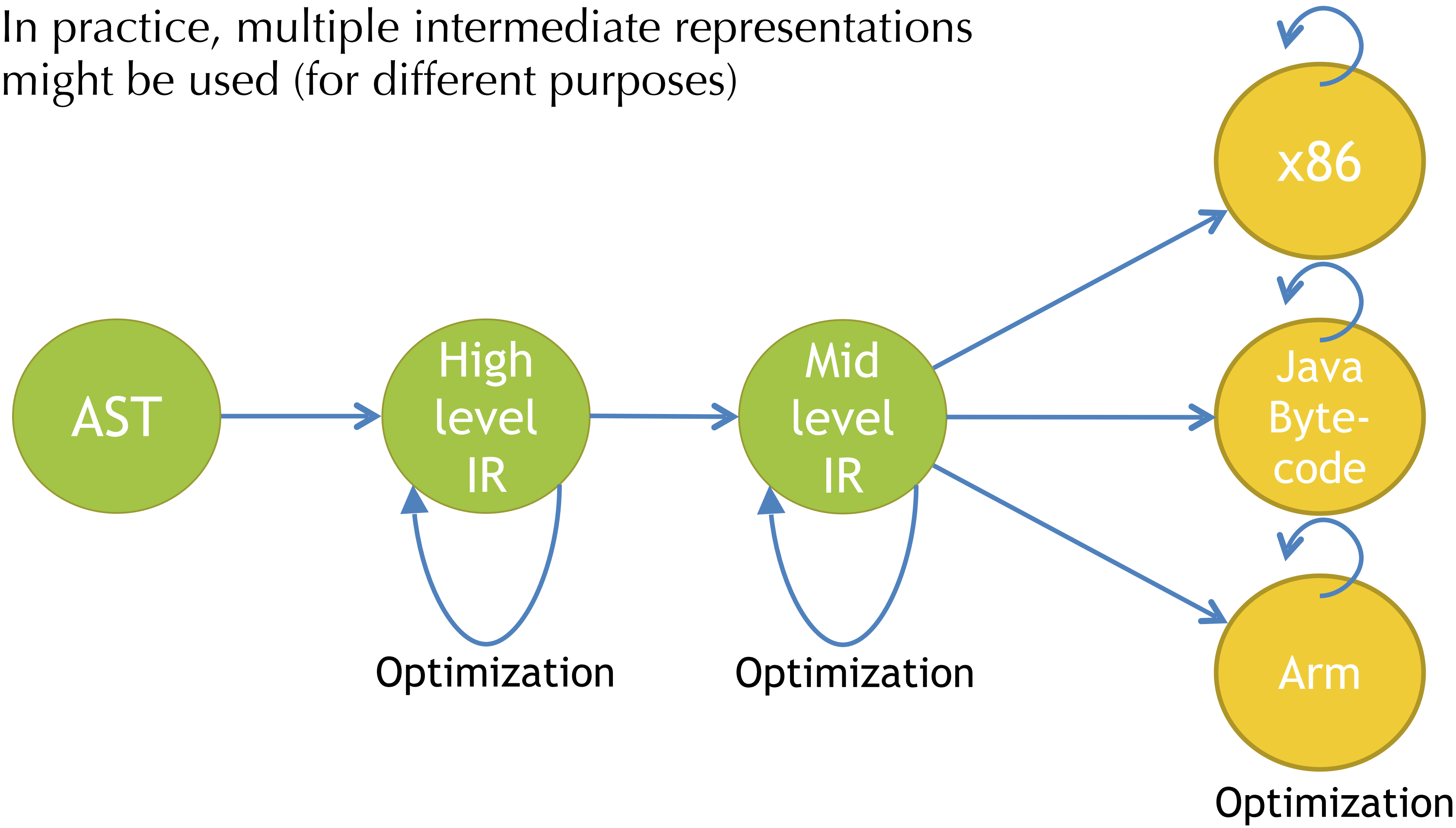
Intermediate Representations (IR's)

- **Abstract machine code**: hides details of the target architecture
- Allows machine independent code generation and optimization.



Multiple IR's

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
- In practice, multiple intermediate representations might be used (for different purposes)



What makes a good IR?

- Easy translation target (from the level above)
- Easy to translate (to the level below)
- Narrow interface
  - Fewer constructs means simpler phases/optimizations

What makes a good IR?

- Easy translation target (from the level above)
- Easy to translate (to the level below)
- Narrow interface
  - Fewer constructs means simpler phases/optimizations

- Example: Source language might have "while", "for", and "foreach" loops (and maybe more variants)
  - IR might have only "while" loops and sequencing
  - Translation eliminates "for" and "foreach"

```
⟦for(pre; cond; post) {body}⟧
              =
 ⟦pre; while(cond) {body;post}⟧
```

  - Here the notation ⟦cmd⟧ denotes the "translation" or "compilation" of the command cmd.

IR's at the extreme

- High-level IR's
  - Abstract syntax + new node types not generated by the parser
    - e.g. Type checking information or disambiguated syntax nodes
  - Typically preserves the high-level language constructs
    - Structured control flow, variable names, methods, functions, etc.
    - May do some simplification (e.g. convert `for` to `while`)
  - Allows high-level optimizations based on program structure
    - e.g. inlining "small" functions, reuse of constants, etc.
  - Useful for semantic analyses like type checking

IR's at the extreme

- High-level IR's
  - Abstract syntax + new node types not generated by the parser
    - e.g. Type checking information or disambiguated syntax nodes
  - Typically preserves the high-level language constructs
    - Structured control flow, variable names, methods, functions, etc.
    - May do some simplification (e.g. convert `for` to `while`)
  - Allows high-level optimizations based on program structure
    - e.g. inlining "small" functions, reuse of constants, etc.
  - Useful for semantic analyses like type checking

- Low-level IR's
  - Machine dependent assembly code + extra pseudo-instructions
    - e.g. a pseudo instruction for interfacing with garbage collector or memory allocator (parts of the language runtime system)
    - e.g. (on x86) a `imulq` instruction that doesn't restrict register usage
  - Source structure of the program is lost:
    - Translation to assembly code is straightforward
  - Allows low-level optimizations based on target architecture
    - e.g. register allocation, instruction selection, memory layout, etc.

IR's at the extreme

- High-level IR's
  - Abstract syntax + new node types not generated by the parser
    - e.g. Type checking information or disambiguated syntax nodes
  - Typically preserves the high-level language constructs
    - Structured control flow, variable names, methods, functions, etc.
    - May do some simplification (e.g. convert `for` to `while`)
  - Allows high-level optimizations based on program structure
    - e.g. inlining "small" functions, reuse of constants, etc.
  - Useful for semantic analyses like type checking

- Low-level IR's
  - Machine dependent assembly code + extra pseudo-instructions
    - e.g. a pseudo instruction for interfacing with garbage collector or memory allocator (parts of the language runtime system)
    - e.g. (on x86) a `imulq` instruction that doesn't restrict register usage
  - Source structure of the program is lost:
    - Translation to assembly code is straightforward
  - Allows low-level optimizations based on target architecture
    - e.g. register allocation, instruction selection, memory layout, etc.

- What's in between?

Mid-level IR's: Many Varieties

- Intermediate between AST (abstract syntax) and assembly
- May have unstructured jumps, abstract registers or memory locations
- Convenient for translation to high-quality machine code
  - Example: all intermediate values might be named to facilitate optimizations that attempt to minimize stack/register usage

- Many examples of Its:
  - Triples: OP a b
    - Useful for instruction selection on X86 via "tiling"
  - Quadruples: a = b OP c ("three address form")
  - SSA: variant of quadruples where each variable is assigned exactly once
    - Easy dataflow analysis for optimization
    - e.g. LLVM: industrial-strength IR, based on SSA
  - Stack-based:
    - Easy to generate
    - e.g. Java Bytecode, UCODE

Growing an IR

- Develop an IR in detail… starting from the very basic.
- Start: a (very) simple intermediate representation for the arithmetic language
  - Very high level
  - No control flow
- Goal: A simple subset of the LLVM IR
  - LLVM = "Low-level Virtual Machine"
  - Used in HW3+
- Add features needed to compile rich source languages