Code Generation

Roadmap

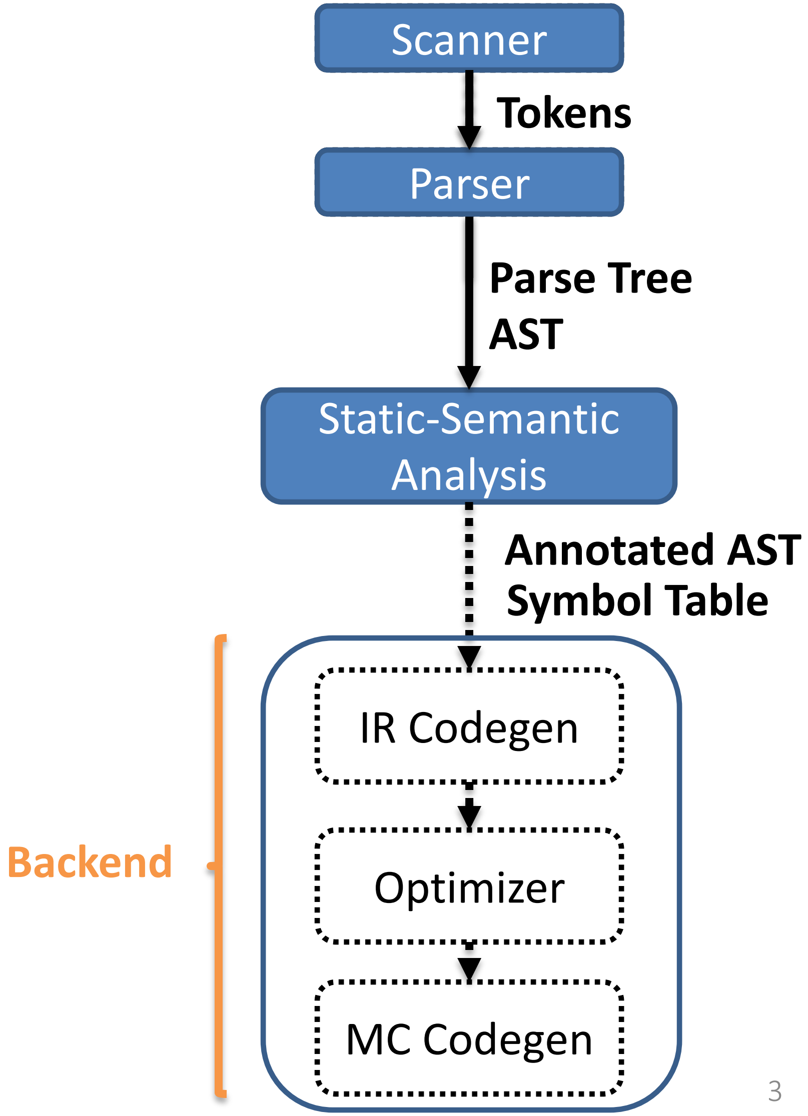
Last time, we learned about variable access

- Local vs. global variables
- Static vs. dynamic scopes

Today

- We'll start getting into the details of MIPS
- Code generation

Roadmap



Scanner
Tokens
Parser
Parse Tree
AST
Static-Semantic Analysis
Annotated AST
Symbol Table
IR Codegen
Optimizer
MC Codegen
Backend

The Compiler Back End

Unlike in the front end, we can skip phases without sacrificing correctness

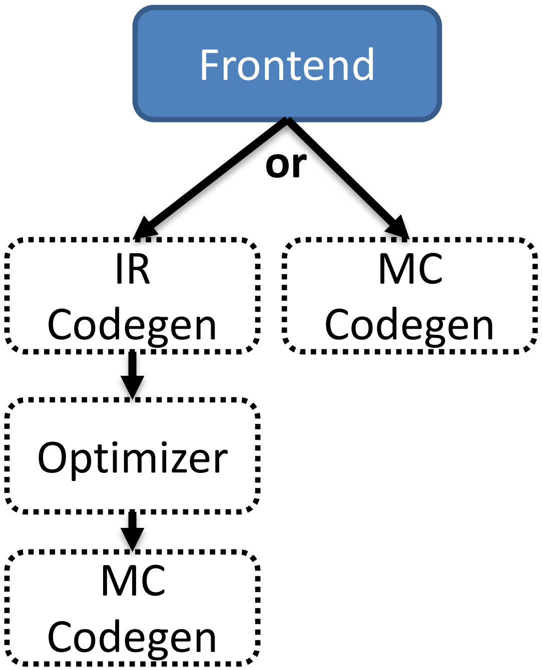
Actually have a couple of options:

- What phases do we do?
- How do we order our phases?

Outline

Possible compiler designs

- Generate IR code or machine-code code directly?
- Generate during SDT or as another phase?



How Many Passes Do We Want?

Fewer passes

- Faster compiling
- Less storage required
- May increase burden on programmer

More passes

- Heavyweight
- Can lead to better modularity

To Generate IR Code or Not?

Generate Intermediate Representation:

- More amenable to optimization
- More flexible output options
- Can reduce the complexity of code generation

Go straight to machine code:

- Much faster to generate code (skip 1 pass, at least)
- Less engineering in the compiler

What Might the IR Do?

Provide illusion of infinitely many registers

“Flatten out” expressions

– Does not allow building up complex expressions

3AC (Three-Address Code)

– Instruction set for a fictional machine

– Every operator has at most 3 operands

3AC Example

```
if  (x + y * z > x * y + z)
    a = 0;
b = 2;
```

```
tmp1 = y * z
tmp2 = x+tmp1
tmp3 = x*y
tmp4 = tmp3+z
if (tmp2 <= tmp4) goto L
   a = 0
L: b = 2
```

3AC Instruction Set

Assignment
- x = y op z
- x = op y
- x = y

Jumps
- if (x op y) goto *L*

Indirection
- x = y[z]
- y[z] = x
- x = &y
- x = *y
- *y = x

Call/Return
- param x,k
- retval x
- call p
- enter p
- leave p
- return
- retrieve x

Type Conversion
- x = AtoB y

Labeling
- label L

Basic Math
- times, plus, etc.

3AC Representation

Each instruction represented using a structure called a “quad”

- Space for the operator
- Space for each operand
- Pointer to auxilary info
  - Label, succesor quad, etc.

Chain of quads sent to an architecture-specific machine-code-generation phase

b: Skip Building a Separate IR

Generate code (of a very simple kind) by traversing the AST

- Add codeGen methods to the AST nodes
- Directly emit corresponding code into file

Correctness/Efficiency Tradeoffs

Two high-level goals

1. Generate correct code
2. Generate *efficient* code

It can be difficult to achieve both of these at the same time

- Why?

A Simplified Strategy

Make sure we don't have to worry about running out of registers

- For each operation (built-in, like plus, or user-defined, like a call on a user-define function), we'll put all arguments on the stack
- We'll make liberal use of the stack for computation
- We'll make use of only two registers
  - Only use $t1 and $t0 for computation

The CodeGen Pass

We’ll now go through a high-level idea of how the topmost nodes in the program are generated

The Responsibility of Different Nodes

Many nodes simply “direct traffic”

- ProgramNode.codeGen
  - call codeGen on the child
- List-node types
  - call codeGen on each element in turn
- DeclNode
  - StructDeclNode – no code to generate!
  - FnDeclNode – generate function body
  - VarDeclNode – varies on context! Globals vs. locals

Generating a Global-Variable Declaration

Source code:

```
int name;
struct MyStruct instance;
```

In varDeclNode

Generate:

```
       .data
       .align 2  #Align on word boundaries
_name: .space N  #(N is the size of variable)
```

Generating a Global-Variable Declaration

```
        .data
        .align 2  #Align on word boundaries
_name: .space N  #(N is the size of variable)
```

How do we know the size?

– For scalars, well-defined: int, bool (4 bytes)

– structs, 4 * size of the struct

We can calculate this during name analysis

Generating Function Definitions

Need to generate

- Preamble
  - Sort of like the function signature
- Prologue
  - Set up the function’s AR
- Body
  - Code to perform the computation
- Epilogue
  - Tear down the function’s AR

MIPS Cra

Also $LO and $HI, special-purpose registers used by multiplication and division instructions

Registers

Register	Purpose
$sp	stack pointer
$fp	frame pointer
$ra	return address
$v0	used for system calls and to return int values from function calls, including the syscall that reads an int
$f0	used to return double values from function calls, including the syscall that reads a double
$a0	used for output of int and string values
$f12	used for output of double values
$t0 - $t7	temporaries for ints
$f0 - $f30	registers for doubles (used in pairs; i.e., use $f0 for the pair $f0, $f1)

Program Structure

For the main function, generate:

```
        .text
        .globl main
main:
```

For all other functions, generate:

```
        .text
_<functionName>:
```

- Data
  - Label: .data
  - Variable names & size; heap st
- Code
  - Label: .text
  - Program instructions
  - Starting location: **main**
  - Ending location

Data

name: type value(s)

– E.g.
  - `v1: .word 10`
  - `a1: .byte ‘a’ , ’b’`
  - `a2: .space 40`
    – 40 here is allocated space – no value is initialized

Memory Instructions

```
lw   register_destination, RAM_source
```

– copy word (4 bytes) at source RAM location to destination register.

```
lb   register_destination, RAM_source
```

– copy byte at source RAM location to low-order byte of destination register

```
li   register_destination, value
```

– load immediate value into destination register

Memory Instructions

```
sw  register_source, RAM_dest
```

– store word in source register into RAM destination

```
sb  register_source, RAM_dest
```

– store byte in source register into RAM destination

Arithmetic Instructions

`add`	`$t0,$t1,$t2`	
`sub`	`$t2,$t3,$t4`	
`addi`	`$t2,$t3, 5`	
`addu`	`$t1,$t6,$t7`	
`subu`	`$t1,$t6,$t7`	
`mult`	`$t3,$t4`	Stores result in $LO
`div`	`$t5,$t6`	Stores result in $LO and Remainder in $HI
`mfhi`	`$t0`	Move from $HI to $t0
`mflo`	`$t1`	Move from $LO to $t1

Control In

Unconditional branch to target

- Specified as a relative transfer of control to target (i.e., target = IP + delta)
- IP implicit; delta is a 16-bit immediate operand (a signed 16-bit number)

```
b       target
beq     $t0,$t1,tar
blt     $t0,$t1,tar
ble     $t0,$t1,tar
bgt     $t0,$t1,targ
bge     $t0,$t1,target
bne     $t0,$t1,
```

Unconditional jump to target

- Specified as an absolute transfer of control to target
- Target limited to 26 bits

Indirect jump

- Specified as an absolute transfer of control to address in $t3

```
j       target
jr      $t3
```

```
jal     sub_label         #  "jump and link"
```

Jump to sub_label, and store the return address in $ra

TODO

MIPS tutorial

– https://minnie.tuhs.org/CompArch/Resources/mips_quick_tutorial.html

Roadmap

Today

- Talked about compiler back-end design points
- Decided to go directly from AST to machine code for our compiler

Next time:

- Run through what the actual codegen pass looks like