Code Generation

Main topic:
1. Two approaches for backend: IR -> Optimization -> MC, or directly to MC.
2. Intermediate representation (IR): still have high level structures, but break complex expressions and introduces labels, jumps, etc.
3. Introduce MIPS instructions.
4. Start talking about CodeGen method on nodes of our AST.

Roadmap

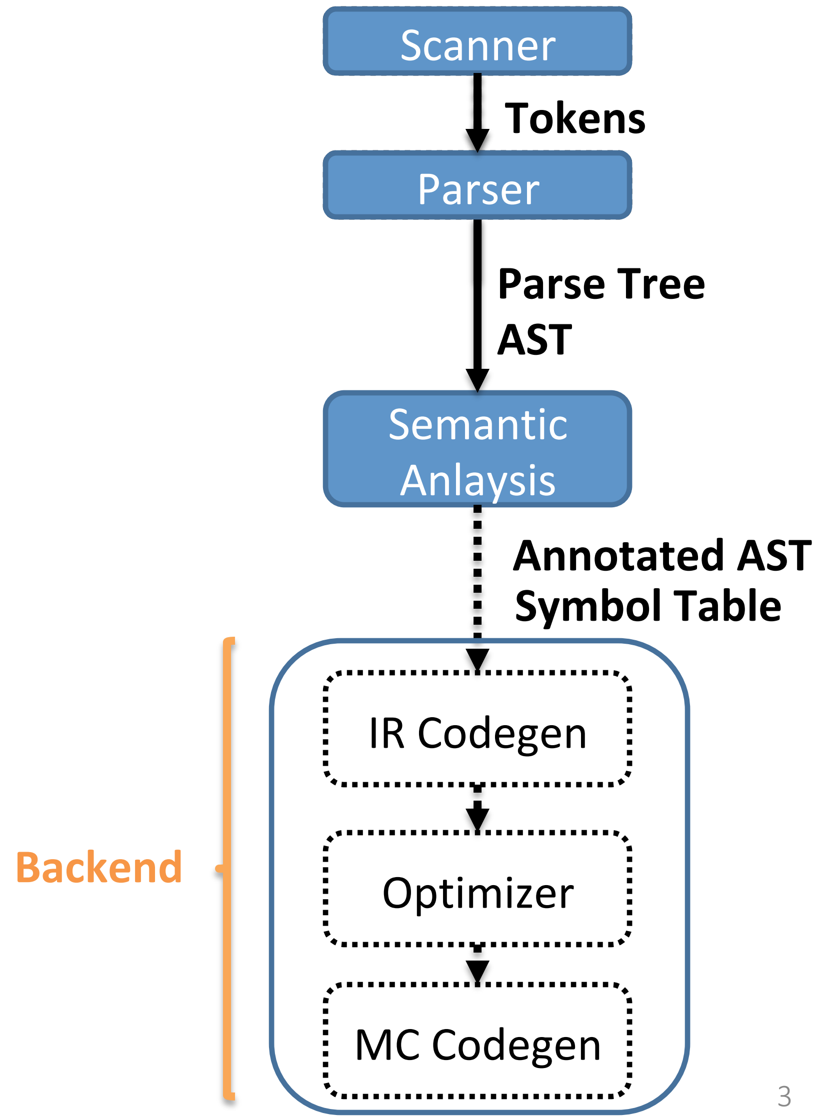
Last time, we learned about variable access

– 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
Semantic Anlaysis
Annotated AST
Symbol Table
Backend
IR Codegen
Optimizer
MC Codegen

The Compiler Back-end

Unlike front-end, we can skip phases without sacrificing correctness

Actually have a couple of options

– What phases do we do

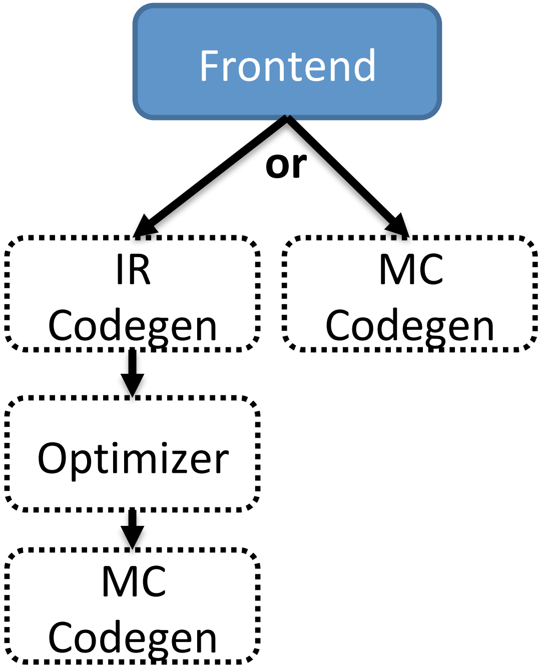
– How do we order our phases

Outline

Possible compiler designs

– Generate IR code or MC code directly?

– Generate during SDT or as another phase?



IR: Intermediate Representation
MC: Machine Code.

Possible considerations when choosing between the two options:
1. Value optimality of the generated machine code, or the speed of compilation?

2. Targeting multiple architecture, or single architecture? Java has intermediate code to achieve cross-platform.

How many passes do we want?

Fewer passes
- Faster compiling
- Less storage requirements
- May increase burden on programmer

More passes
- Heavyweight
- Can lead to better modularity
- We'll go with this approach for our language

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 build-up of complex expressions

3AC (Three-Address Code)

– Pseudocode-machine style instruction set

– Every operator has at most 3 operands

3AC Example

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

Two operations:
1. Break down complex expression into binary operations;
2. Negate if statements into goto.

```
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

IR still has some high level code and looks like pseudo code. Though it breaks complex expressions and introduces labels.

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

3AC LLVM Example

Demo

Optimization is done using IR, instead of MC. The reason is that IR still maintains the structure of the original code, and it's easier to do optimization on it.

To do optimization, multiple passes are gone through the IR.

Direct machine code generation

Option 1

- Have a chain of quad-like structures where each element is a machine-code instruction
- Pass the chain to a phase that writes to file

Option 2

- Write code directly to the file
- Greatly aided by assembly conventions here
- Assembler allows us to use function names, labels in output

Our language: skip the IR

Traverse AST

- Add codeGen methods to the AST nodes
- Directly write 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?

Simplifying assumptions

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

- We'll put all function arguments on the stack
- We'll make liberal use of the stack for computation
    - 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 Effect of Different Nodes

Many nodes simply structure their results

- ProgramNode.codeGen
  - call codeGen on the child
- List node types (e.g., StmtList)
  - call codeGen on each element in turn
- DeclNode
  - StructDeclNode – no code to generate!
  - FnDeclNode – generate function body
  - VarDeclNode – varies on context! Globals v locals

Doesn't exist at the runtime. The information is in the symbol table.

Global Variable Declarations

Source code:

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

In varDeclNode

Generate:

```
        .data
        .align 4  #Align on word boundaries
_name: .space N  #(N is the size of variable)
```

Generating Global Variable Declaration

```
         .data
         .align 4  #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

Create an entry point to the function using a label, etc.

- Prologue
  - Set up the function

Responsibility for entering the function, like setting up return address, setting up parameters, etc.

- Body
  - Perform the computation

Call BodyNode.CodeGen.

- Epilogue
  - Tear down the function

Responsibility for exiting the function, like putting return value on the stack, restoring $fp of the caller, etc.

MIPS crash course

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

Data

- Label: .data
- Variable names & size; heap storage

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

Mem 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

Mem 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
mflo    $t1
```

mf means "move from"

Control instructions

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

```
j         target
jr        $t3
```

r: register

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

Jump and store return address in $31

TODO

Watch ***ALL*** MIPS and SPIM tutorials online

- https://www.youtube.com/playlist?list=PLYbkk0SELNlwTDg1LISzYwunPWqflEkM4

MIPS tutorial

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

Roadmap

Today

- Talked about compiler backend design points
- Decided to go with direct to machine code design for our language

Next time:

- Run through what actual codegen pass will look like