CS 536

Code Generation

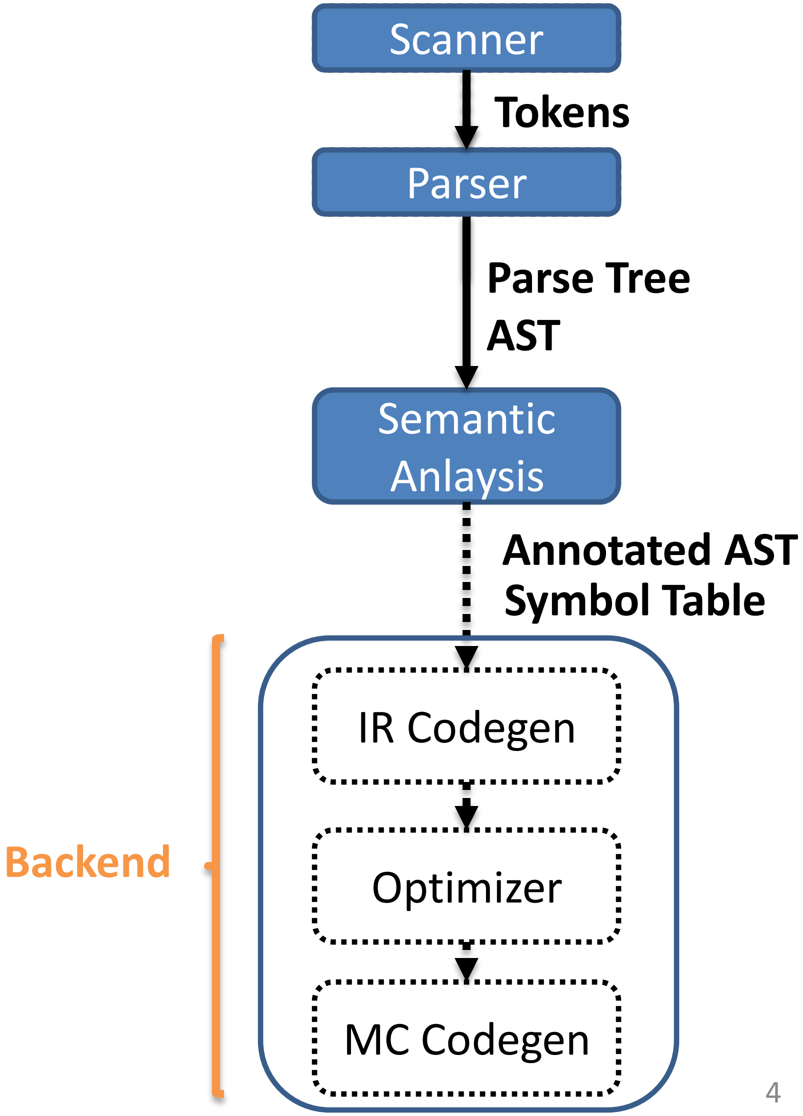
Project 5 Oracle

davidson-www.cs.wisc.edu/p5_oracle.php

Homework 8

- Should be posted now
- Due next Tuesday

Roadmap

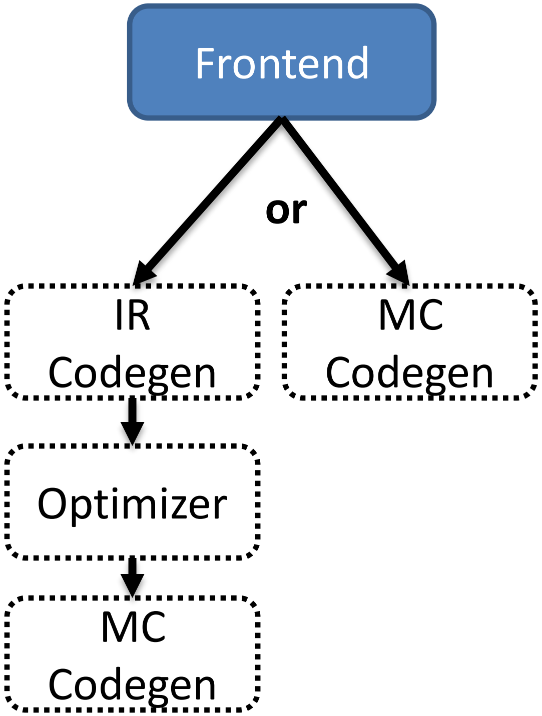

Scanner
Tokens
Parser
Parse Tree
AST
Semantic Anlaysis
Annotated AST
Symbol Table
IR Codegen
Optimizer
MC Codegen
Backend

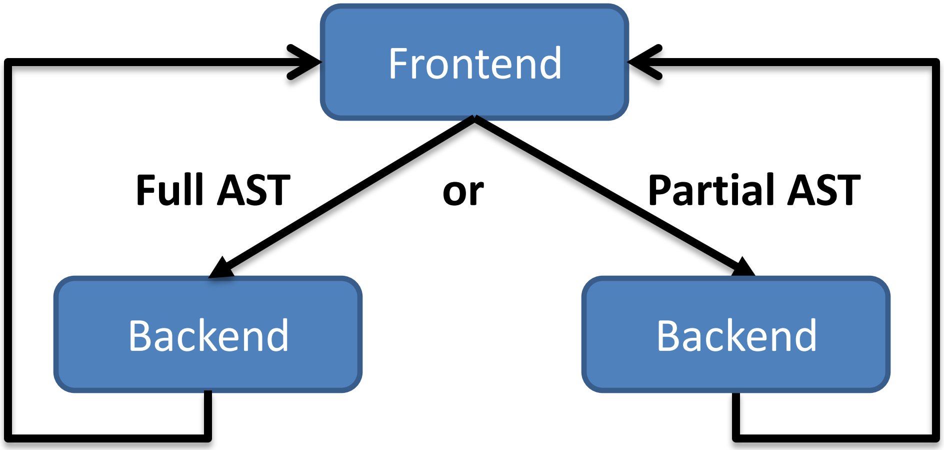
The Compiler Back-end

- 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?





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 C-Flat

To Generate IR Code or Not?

- If we do generate an Intermediate Representation:
  - More amenable to optimization
  - More flexible output options
  - Can reduce the complexity of code generation
- If we 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?

- Infinite-register operations
- “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;
```

```
   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 MC codegen phase

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

C-Flat: Skip the IR

- Traverse AST
  - add codeGen methods to the AST nodes
  - Directly spit 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 the 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
    - call codeGen on each element in turn
  - DeclNode
    - StructDeclNode – no code to generate!
    - FnDeclNode – generate function body
    - VarDeclNode – varies on context! Globals v locals

Generating 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 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
  - Body
    - Do the thing
  - Epilogue
    - Tear down the function

Next time

- Actual MIPS code that the nodes will generate