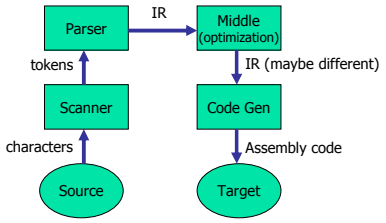
# Code Generation

CSE413
Autumn 2007



# Compiler Structure (review)






# What We Need

- To run a D program:
  - Space needs to be allocated for a stack and a heap
  - ESP and other registers need to have sensible initial values
  - We need some way to communicate with the outside world (I/O)



# Bootstraping from C

- **Idea**: take advantage of the existing C runtime library
- Write a small C main program that calls the main method in the assembly you generate as if it were a C function.
  - C's standard library provides the execution environment
  - We can write C functions for get() and put() and call them from our asm code.



# Bootstrap Program

- The bootstrap will be a tiny C program that calls your compiled code as if it were an ordinary C function
- It also contains some functions that compiled code can call as needed
  - Mini "runtime library"
  - get(), put()



# Example Bootstrap Program

```
#include <stdio.h>
int d$main();   /* prototype for external function */

int main() {
  printf("\nValue returned from D main: %d.\n", d$main());
  return 0;
}

/* return next integer from standard input */
int get() { scanf.... }
/* write x to standard output */
int put(int x) { printf... }
```

## Code Generation in Our Project



## Generating .asm Code

- **Suggestion**: isolate the actual output operations in a handful of routines
  - Modularity & saves some typing
  - Possibilities

```
// write code string s to .asm output
void gen(String s) { … }
// write "op  src,dst" to .asm output
void genbin(String op, String src, String dst) { … }
// write label L to .asm output as "L:"
void genLabel(String L) { … }
```

  - A handful of these methods should do it



## Agenda

- Mapping source code to x86
  - **Today**: Stuff Needed for project: basic statements and expressions
  - **Next**: Other cool stuff: Object representation, method calls, and dynamic dispatch



## Conventions for Examples

- Examples show code snippets in isolation
- A "real" code generator needs to worry about things like :
  - Which registers are busy at which point in the program
  - Which registers to spill into memory when a new register is needed and no free ones are available
  - You won't need to worry about this for your compiler!
- Register eax used below as a generic example
  - Rename as needed for more complex code involving multiple registers
  - But for the most part in our compiler we can stick to just using eax and ecx.



## A Simple Code Generation Strategy

- Priority: quick 'n dirty correct code first, optimize later if time
- Treat the x86 as a 1-register stack machine



## x86 as a Stack Machine

- **Idea**: Use x86 stack for expression evaluation with eax as the "top" of the stack
- **Invariant**: Whenever an expression (or part of one) is evaluated at runtime, the result is in eax
- If a value needs to be preserved while another expression is evaluated, push eax, evaluate, then pop when needed
  - Remember: always pop what you push
  - Will produce lots of redundant, but correct, code

## Constants

- Source
  17
- x86
  mov eax, 17
  - Idea: realize constant value in a register
- Aside: Optimization: if constant is 0
  xor eax, eax



## Use (RHS) of Variables

- Source
  (use of variable) a
- x86
  mov eax, [ebp+ -8]
  - All variables in our programs will be addressable relative to ebp.
  - In this example a is a *local variable*.
  - If the offset was positive, it would be a *parameter*.
  - Check your symbol table to generate the correct offset for a given variable.



## Assign (LHS) of Variables

- Source
  a = exp
- x86
  <code for exp, leaves result in eax>
  mov [ebp+ -8], eax



## Example: var = exp;

- Assuming that var is a local variable
  - <code for exp>
    - Generates code that leaves the result of evaluating exp in eax
  - gen(mov [ebp+offset of variable],eax)



## Example: Generate Code for Constants and Identifiers

- Integer constants, say 17
  gen(mov eax,17)
  - leaves value in eax
- Variables
  gen(mov eax, [ebp + appropriate offset])
  - also leaves value in eax



## Assignment Statement

- Source
  var = exp;
- x86
  <code to evaluate exp,
  leaving result in register eax>
  mov [ebp+$offset_{var}$], eax

## Binary +

- Source

  exp1 + exp2

- x86

  <code evaluating exp1 into eax>
  <code evaluating exp2 into ecx>
  add eax, ecx



## Example: Generate Code for exp1 + exp2

- <code to calculate exp1>
  - generates code to evaluate exp1 and put result in eax
- gen(push eax)
  - generate a push instruction
- <code to calculate exp2>
  - generates code for exp2; result in eax
- gen(pop ecx)
  - pop left argument into ecx; cleans up stack
- gen(add eax, ecx)
  - perform the addition; result in eax



## Binary -, *

- Same as +
  - Use sub for –
    - // Be sure to get order of operands correct!!
  - Use imul for *
- Aside: Optimizations
  - Use left shift to multiply by powers of 2
  - Use x+x instead of 2*x, etc. (faster)
  - Use dec for x-1



## Control Flow

- Basic idea: decompose higher level operation into conditional and unconditional gotos
- In the following, $j_{false}$ is used to mean: "jump when a condition is false"
  - No such instruction on x86
  - Will have to realize with appropriate sequence of instructions to set condition codes followed by conditional jumps
  - Normally won't actually generate the value "true" or "false" in a register



## While

- Source

  while (cond) stmt

- x86

  test: <code evaluating cond>
  $j_{false}$ done
  <code for stmt>
  jmp test
  done:



## Labels

- In x86 assembly language we'll need to produce unique labels for each if, while, etc.
- Labels can appear on a line by themselves.

  jmp label

  - will start execution on the first line of x86 code it finds after seeing the label.

## Example: Control Flow: Unique Labels

- Needed: a String-valued method that returns a different label each time it is called (e.g., L1, L2, L3, …)
  - Variation: a set of methods that generate different kinds of labels for different constructs (can really help readability of the generated code)
    - (while1, while2, while3, …; if1, if2, …; else1, else2, ….)



## If

- Source

  if (cond) stmt
- x86

  <code evaluating cond>
  $j_{false}$ skip
  <code for stmt>
  skip:



## If-Else

- Source

  if (cond) stmt1 else stmt2
- x86

  <code evaluating cond>
  $j_{false}$ else
  <code for stmt1>
  jmp done
  else: <code for stmt2>
  done:



## Boolean Expressions

- What do we do with this?

  x > y
- It is an expression that evaluates to true or false
  - Could generate the value (0/1 or whatever the local convention is)
  - But normally we don't want/need the value; we're only trying to decide whether to jump



## Code for exp1 > exp2

- Basic idea: designate jump target, and whether to jump if the condition is true or if it is false
- Example: exp1 > exp2, target L123, jump on false

  <evaluate exp1 to eax>
  <evaluate exp2 to ecx>
  cmp eax, ecx
  jng L123



## Example exp1 < exp2

- Similar to other binary operators
- Difference: context is a target label and whether to jump if true or false
- x86:

  <evaluate exp1 to eax>
  gen(push eax)
  <evaluate exp2 to eax>// eax contains exp2
  gen(pop ecx) // ecx contains exp1
  gen(cmp ecx, eax)
  gen(condjump targetLabel)
  - appropriate conditional jump depending on sense of test

## Boolean Operators: !

- Source
  ! bool-exp
- Context: evaluate bool-exp and jump to L123 if false (or true)
- To compile !, reverse the sense of the test: evaluate bool-exp and jump to L123 if true (or false)



## Example: foo (exp1, exp2)

```
<evaluate exp1; result in eax>
push   eax          ; push parameter
< evaluate exp2; result in eax>
push   eax          ; push parameter
call   foo          ; call external put routine
add    esp, 8       ; pop parameters
```

- For our compiler we will push parameters from left to right (this deviates from the standard calling convention but will make code generation slightly easier)



## Example: put(exp)

```
< evaluate exp; result in eax>
push   eax          ; push parameter
call   _put         ; call external put routine
add    esp,4        ; pop parameter
```

- Calls to get and put routines must have their name pre-pended with an underscore
- Otherwise compile like any other function!



## Function Definitions

- Generate label for function
- Generate function prologue
- Generate code for statements in order
  - Method epilogue will be generated as part of each return statement (next)



## Example: Function Definition

int foo(int a, int b) { ...

- x86

```
gen(function_label:);     // generate label for function body
gen(push ebp);            // save ebp
gen(mov ebp, esp);        // set up frame pointer
gen(sub esp, #_bytes_for_locals);  // allocate space for
                                   // local variables

<generated code for function body follows>
```



## Example: return exp;

Generate method epilogue to unwind the stack frame; end with ret instruction

- x86

```
<code for exp, leaving result in eax>
gen(mov  esp, ebp)     // free local vars
gen(pop ebp)
gen(ret)
```