



# CSE P 501 – Compilers

---

Running MiniJava  
Basic Code Generation and Bootstrapping  
Hal Perkins  
Autumn 2009



# Agenda

---

- Enough to get a working project
  - Assembler source file format
  - A very basic code generation strategy
  - Interfacing with the bootstrap program
  - Implementing the system interface



# What We Need

---

- To run a MiniJava 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 allocate storage (new) and communicate with the outside world



# Bootstrapping from C

---

- Idea: take advantage of the existing C runtime library
- Use a small C main program to call the MiniJava main method as if it were a C function
- C's standard library provides the execution environment and we can call C functions from compiled code for I/O, malloc, etc.



# Assembler File Format

---

- Here is a skeleton for the .asm file to be produced by MiniJava compilers (MASM syntax)

```
.386                ; use 386 extensions
.model flat,c       ; use 32-bit flat address space with
                    ; C linkage conventions for
                    ; external labels

public asm_main     ; start of compiled static main
extern put:near,get:near,mjmalloc:near ; external C routines
.code
;; generated code   ] repeat .code/.data as needed
.data
;; generated method tables ]
...
end
```



# GNU Assembler File Format

---

- GNU syntax is roughly the same

```
.text                # code segment
.globl asm_main     # start of compiled static main
;; generated code   repeat .code/.data as needed
.data
;; generated method tables # repeat .text/.data as needed
...
end
```



# External Names

---

- In a unix environment, an external symbol is used as-is
- In Windows, the convention is that an external symbol `xyzy` appears in the asm code as `_xyzy` (leading underscore)
  - True in both VS `masm` and `gnu assembler` under `cygwin`
  - Also true on Intel OS X systems?
- You should adapt to whatever environment you're using



# Intel vs. GNU Syntax

- The GNU assembler uses AT&T syntax for historical reasons. Main differences:

	Intel/Microsoft	AT&T/GNU as
Operand order: op a,b	a = a op b (dst first)	b = a op b (dst last)
Memory address	[baseregister+offset]	offset(baseregister)
Instruction mnemonics	mov, add, push, ...	movl, addl, pushl [operand size is added to end]
Register names	eax, ebx, ebp, esp, ...	%eax, %ebx, %ebp, %esp, ...
Constants	17, 42	\$17, \$42
Comments	; to end of line	# to end of line or /* ... */





# Generating .asm Code

---

- Suggestion: isolate the actual compiler 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

# A Simple Code Generation Strategy



---

- Goal: quick 'n dirty correct code, optimize later if time
- Traverse AST primarily in execution order and emit code during the traversal
  - May need to control the traversal from inside the visitor methods, or have both bottom-up and top-down visitors
- Treat the x86 as a 1-register stack machine at first
  
- Alternative strategy: produce lower-level linear IR and generate from that (after possible optimizations)
  - Usually more ambitious than is reasonable for 10 weeks



# x86 as a Stack Machine

---

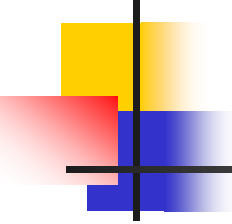
- Idea: Use x86 stack for expression evaluation with `eax` as the “top” of the stack
- Invariant: Whenever an expression (or part of one) is evaluated at runtime, the result is in `eax`
- If a value needs to be preserved while another expression is evaluated, push `eax`, evaluate, then pop when needed
  - Remember: always pop what you push
  - Will produce lots of redundant, but correct, code
- Examples below follow code shape examples, but with some details about where code generation fits



# Example: Generate Code for Constants and Identifiers

---

- Integer constants, say 17
  - gen(mov eax,17)
    - leaves value in eax
- Variables (whether int, boolean, or reference type)
  - gen(mov eax,[appropriate base register+ appropriate offset])
    - also leaves value in eax



# Example: Generate Code for $exp1 + exp1$

---

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



# Example: `var = exp;` (1)

---

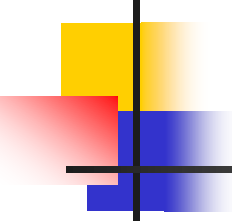
- Assuming that `var` is a local variable
  - visit node for `exp`
    - Generates code that leaves the result of evaluating `exp` in `eax`
  - `gen(mov [ebp+offset of variable],eax)`



## Example: `var = exp;` (2)

---

- If `var` is a more complex expression (object or array reference, for example)
  - visit `var`
  - `gen(push eax)`
    - push reference to variable or object containing variable onto stack
  - visit `exp`
  - `gen(pop edx)`
  - `gen(mov [edx+appropriate_offset],eax)`



# Example: Generate Code for `obj.f(e1,e2,...en)`

---

- Visit `en`
  - leaves argument in `eax`
- `gen(push eax)`
- ... Repeat until all arguments pushed
- Visit `obj`
  - leaves reference to object in `eax`
  - Note: this isn't quite right if evaluating `obj` has side effects – ignore for simplicity for now
- `gen(mov ecx, eax)`
  - copy "this" pointer to `ecx`
- generate code to load method table pointer
- generate call instruction with indirect jump
- `gen(add esp, numberOfBytesOfArguments)`
  - Pop arguments





# Method Definitions

---

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



## Example: return exp;

---

- Visit exp; leaves result in eax where it should be
- Generate method epilogue to unwind the stack frame; end with ret instruction



# 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, ...; fi1, fi2, ... .)



# Control Flow: Tests

---

- Recall that the context for compiling a boolean expression is
  - Jump target
  - Whether to jump if true or false
- So visitor for a boolean expression needs this information from parent node



# Example: while(exp) body

---

- Assuming we want the test at the bottom of the generated loop...
  - gen(jmp testLabel)
  - gen(bodyLabel:)
  - visit body
  - gen(testLabel:)
  - visit exp (condition) with target=bodyLabel and sense="jump if true"



# Example $exp1 < exp2$

---

- Similar to other binary operators
- Difference: context is a target label and whether to jump if true or false
- Code
  - visit exp1
  - gen(push eax)
  - visit exp2
  - gen(pop edx)
  - gen(cmp eax,edx)
  - gen(condjump targetLabel)
    - appropriate conditional jump depending on sense of test



# Boolean Operators

---

- `&&` and `||`
  - Create label needed to skip around second operand when appropriate
  - Generate subexpressions with appropriate target labels and conditions
- `!exp`
  - Generate `exp` with same target label, but reverse the sense of the condition



# Join Points

---

- Loops and conditional statements have join points where execution paths merge
- Generated code must ensure that machine state will be consistent regardless of which path is taken to reach a join point
  - i.e., the paths through an if-else statement must not leave a different number of bytes pushed onto the stack
  - If we want a particular value in a particular register at a join point, both paths must put it there, or we need to generate additional code to get value in the right register
- With a simple 1-accumulator model of code generation, this should generally be true without needing extra work; with better use of registers this becomes an issue





# 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”
  - You can add to this if you like
    - Sometimes simpler to generate a call to a newly written library routine instead of generating in-line code – implementor tradeoff



# Example Bootstrap Program

---

```
#include <stdio.h>
extern void asm_main(); /* compiled code */
/* execute compiled program */
void main() { asm_main(); }
/* return next integer from standard input */
int get() { ... }
/* write x to standard output */
void put(int x) { ... }
/* return a pointer to a block of memory at least nBytes
   large (or null if insufficient memory available) */
void * runtimealloc(int nBytes) { return malloc(nBytes); }
```



# Interfacing to External Code

---

- Recall that the .asm file includes these declarations at the top

```
public asm_main    ; start of compiled static main
extern put:near,get:near,mjmalloc:near
                  ; external C routines
```

- “public” means that the label is defined in the .asm file and can be linked from external files
  - Jargon: also known as an entry point
- “extern” declares labels used in the .asm file that must be found in another file at link time
  - “near” means in same segment (as opposed to multi-segment MS-DOS programs of ancient times)



# Main Program Label

---

- Compiler needs special handling for the static main method
  - Label must be the same as the one declared extern in the C bootstrap program and declared public in the .asm file
  - `asm_main` used above
    - Can be changed if you wish
    - Why not "main"? (Hint: what is / where is the *real* main function?)



# Interfacing to “Library” code

---

- To call “behind the scenes” library routines:
  - Must be declared extern in generated code
  - Call using normal C language conventions



# System.out.println(exp)

---

- Can handle in an ad-hoc way
  - (particularly since this is a “reserved word” in MiniJava)  
<compile exp; result in eax>  
push    eax                   ; push parameter  
call     put                   ; call external put routine  
add      esp,4                 ; pop parameter
- A more general solution if System.out were a real class:
  - Hand-code (in asm) classes to act as a bridge between compiled code and the C runtime
  - Put information about these classes in the symbol table at compiler initialization
  - Calls to these routines compile normally – no other special case code needed in the compiler(!)



# And That's It...

---

- We've now got enough on the table to complete the compiler project
- Coming Attractions
  - Lower-level IR
  - Back end (instruction selection and scheduling, register allocation)
  - Middle (optimizations)