

CSE P 501 – Compilers

Running MiniJava
Basic Code Generation and Bootstrapping
Hal Perkins
Summer 2004

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-1

Agenda

- What we need to finish the project
 - Assembler source file format
 - A basic code generation strategy
 - Interfacing with the bootstrap program
 - Implementing the system interface

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-2

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 and communicate with the outside world

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-3

Bootstrapping from C

- Idea: take advantage of the existing C runtime library
- Write a small C main program that calls 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.

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-4

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
```

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-5

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

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-6

A Simple Code Generation Strategy

- Priority: quick 'n dirty correct code first, 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
- Alternative strategy: produce lower-level linear IR and generate from that (after possible optimizations)
 - We'll cover this in lecture, but may be more ambitious than we can do in 10 weeks

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-7

x86 as a Stack Machine

- Idea: Use x86 stack for expression evaluation with `eax` as the "top" of the stack
- 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

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-8

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`

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-9

Example: Generate Code for `exp1 + exp2`

- 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`

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-10

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)`

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-11

Example: `var = exp;` (2)

- If `var` is a more complex expression
 - 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)`

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-12

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

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-13

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)

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-14

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

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-15

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, ...)

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-16

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

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-17

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"

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-18

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

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-19

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

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-20

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 the simple 1-accumulator model of code generation, this should generally be true without needing extra work

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-21

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

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-22

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); }
```

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-23

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)

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-24

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 the real main function here?)

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-25

Interfacing to "Library" code

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

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-26

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
 - 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(!)

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-27

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)

11/8/2005

© 2002-05 Hal Perkins & UW CSE

M-28