


CSE 582 – Compilers

Running JFlat
Basic Code Generation and Bootstrapping
Hal Perkins
Autumn 2002


10/14/2002 © 2002 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 JFSystem interface class

10/14/2002 © 2002 Hal Perkins & UW CSE M-2




Assembler File Format

- Here is a skeleton for the .asm file to be produced by JFlat compilers

```

.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 get:near,put:near,jfmalloc:near ; external C routines
.code
;; generated code   } repeat .code/.data as needed
.data
;; generated method tables
...
end
  
```


10/14/2002 © 2002 Hal Perkins & UW CSE M-3



Writing .asm Code

- Suggestion: isolate the actual write 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


10/14/2002 © 2002 Hal Perkins & UW CSE M-4



A Simple Code Generation Strategy

- Priority: produce correct code first, optimize later
- Traverse AST primarily in execution order and emit code during the traversal
 - Will need to control the traversal order for at least some nodes from inside the visitor methods instead of always using the default depth-first order
- 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 too ambitious for the project at this point

10/14/2002 © 2002 Hal Perkins & UW CSE M-5



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

10/14/2002 © 2002 Hal Perkins & UW CSE M-6

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`

10/14/2002

© 2002 Hal Perkins & UW CSE

M-7

Example: Generate Code for $exp1 + exp2$

- Visit `exp1`
 - generate code to evaluate `exp1` and put result in `eax`
- `gen(push eax)`
 - generate a push instruction
- Visit `exp2`
 - generate 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`

10/14/2002

© 2002 Hal Perkins & UW CSE

M-8

Example: `var = exp; (1)`

- Assuming that `var` is a local variable
 - visit node for `exp`
 - `gen(mov [ebp+offset of variable],eax)`

10/14/2002

© 2002 Hal Perkins & UW CSE

M-9

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+appropriateoffset],eax)`

10/14/2002

© 2002 Hal Perkins & UW CSE

M-10

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
- `gen(mov ecx,ecx)`
 - copy "this" pointer to `ecx`
- generate code to load method table pointer
- generate call instruction with indirect jump
- `gen(add esp,numberOfBytesOfArguments)`
 - Pop arguments

10/14/2002

© 2002 Hal Perkins & UW CSE

M-11

Method Definitions

- Generate label for method
- Generate method prologue
- Visit statements in order

10/14/2002

© 2002 Hal Perkins & UW CSE

M-12

Example: return exp;

- Visit exp; leaves result in eax where it should be
- Generate method epilogue ending with ret instruction

10/14/2002 © 2002 Hal Perkins & UW CSE M-13

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 (might help readability of the generated code)
 - (while1, while2, while3, ...; else1, else2, ...)

10/14/2002 © 2002 Hal Perkins & UW CSE M-14

Control Flow: Tests

- Recall that the context for 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

10/14/2002 © 2002 Hal Perkins & UW CSE M-15

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 with target=bodyLabel and sense="jump if true"

10/14/2002 © 2002 Hal Perkins & UW CSE M-16

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)

10/14/2002 © 2002 Hal Perkins & UW CSE M-17

Boolean Operators

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

10/14/2002 © 2002 Hal Perkins & UW CSE M-18

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
 - With our simple model of code generation, this should generally be true without needing extra work

10/14/2002

© 2002 Hal Perkins & UW CSE

M-19

Bootstrap Program

- The bootstrap is 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

10/14/2002

© 2002 Hal Perkins & UW CSE

M-20

Bootstrap Program Code

```
#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 * jfmalloc(int nBytes) { return malloc(nBytes); }
```

10/14/2002

© 2002 Hal Perkins & UW CSE

M-21

Interfacing to External Code

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

```
public asm_main          ; start of compiled static main
extern get:near,put:near,jfmalloc: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

10/14/2002

© 2002 Hal Perkins & UW CSE

M-22

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 have a reason to do so

10/14/2002

© 2002 Hal Perkins & UW CSE

M-23

Interfacing to "Library" code

- To call "behind the scenes" library routines:
 - Must be declared extern in generated code
 - Call using normal C language conventions
- Get and put are different
 - Should be usable as normal methods in JFlat source code

10/14/2002

© 2002 Hal Perkins & UW CSE

M-24

Predefined JFSystem Class

- This class acts as if it had this definition

```
public class JFSystem {
    public JFSystem() { super(); }
    // return next number from input
    public int get(); { ... }
    // write x to output
    public void put(int x) { ... }
}
```

10/14/2002

© 2002 Hal Perkins & UW CSE

M-25

Using JFSystem

- A JFlat program uses JFSystem like this

```
class Main {
    public static void main() {
        JFSystem sys = new JFSystem();
        int k = sys.get();
        sys.put(2*k);
    }
}
```

10/14/2002

© 2002 Hal Perkins & UW CSE

M-26

Implications (1)

- At compile time:
 - Symbol table entries for class JFSystem and its methods need to be created by hand, which allows...
 - Code that uses JFSystem and its methods can be compiled exactly the same other JFlat code
 - No special cases for this class
 - Can be extended(!), etc.

10/14/2002

© 2002 Hal Perkins & UW CSE

M-27

Implications (2)

- At run time:
 - Need appropriate method dispatch tables and "methods" for this class so the generic method call code works without change
 - These are generated by the compiler in the .asm file
 - Observation: when get or put is called, we can redirect this to the library function immediately since necessary arguments and return address for the real get/put are already on the stack(!)

10/14/2002

© 2002 Hal Perkins & UW CSE

M-28

JFSystem Runtime Representation – in .asm Code

```
;; JFSystem method dispatch table
.data
JFSystem$$ dd 0 ; no parent class
            dd JFSystem$JFSystem ; constructor
            dd JFSystem$get ; methods
            dd JFSystem$put
;; "methods" for class JFSystem
.code
JFSystem$JFSystem: ret ; nothing to construct
JFSystem$get: jmp get ; jump to external C
JFSystem$put: jmp put
...
```

10/14/2002

© 2002 Hal Perkins & UW CSE

M-29

And That's It...

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

10/14/2002

© 2002 Hal Perkins & UW CSE

M-30