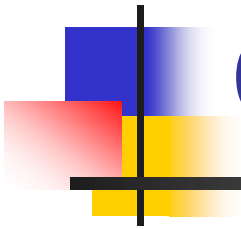
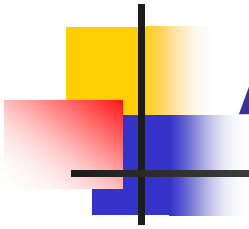


CSE 401 – Compilers

Running MiniJava
Basic Code Generation and Bootstrapping

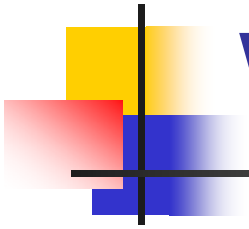
Hal Perkins

Winter 2010



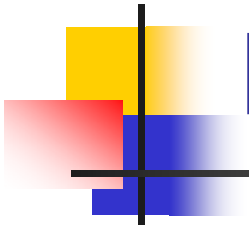
Agenda

- Enough to get a working project
  - Assembler source file format
  - Interfacing with the bootstrap program & outside world
  - A basic code generation strategy



What We Need

- To run a MiniJava program:
  - Space needs to be allocated for a stack and a heap
  - ESP and EBP need to have sensible initial values
  - We need some way to allocate storage and communicate with the outside world



Bootstrapping from C

- Idea: Start execution in a small `main` function written in C
- C `main` calls the compiled MiniJava main method using standard C linkage
- MiniJava's main executes from there
- Compiled code can call back to other functions included in the same C file (malloc, print, ...)
  - Add to this file if you like
    - Sometimes easier for generated code to call an external function than producing the whole thing in-line

Bootstrap Program Sketch

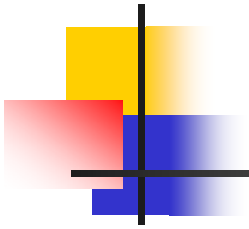
```
#include <stdio.h>
extern void asm_main();  /* compiled code */
/* execute compiled program */
  int main() { asm_main(); return 0; }
/* write x to standard output */
  void put(int x) { printf("...", x); }
/* return a pointer to a block of memory with at least n
   bytes (or null if insufficient memory available) */
  void* runtimealloc(int n) { return malloc(n); }
```

- Actual code is file boot.c linked from codegen project page

GNU Assembler File Format

- Here is a skeleton for the .asm file to be produced by MiniJava compilers (gnu assembler format)

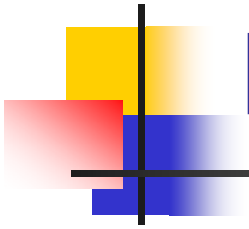
```
        .text                   # code segment
        .globl   asm_main       # declare asm_main as entry point

asm_main:
        # main program starts execution here
        ...

        .data                   # data segment
        # generated method tables & static data
        ...
        # repeat .text/.data as needed
```



Intel vs. GNU Syntax

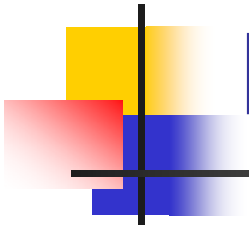
- The GNU assembler uses AT&T syntax for historical reasons. Main differences:

	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 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 /* ... */



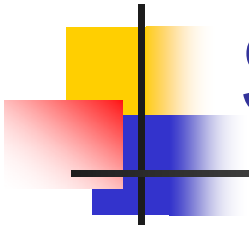
Main Program Label

- Compiler needs special handling for the static main method label
  - Label declared extern in C bootstrap program must match .globl label in the compiler-generates assembly file
  - “asm_main” used in starter code
    - Can’t be “main”. Why not?



External Names (technicality)

- In linux an external symbol is used as-is
- In Windows and Intel OS X, the convention is that an external symbol xyzzy appears in the asm code as _xyzzy (leading underscore)
- Adapt to whatever environment you're using
  - But what you turn in needs to run on attu (linux)

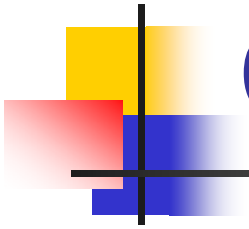


System.out.println(exp)

- Evaluate exp, then call the external put function in boot.c (which calls printf)

```
<compile exp; result in eax>
pushl   %eax          # push exp value
call    put           # call external put routine
addl    $4,%esp       # pop parameter
```

- More sample code in demo.s file linked from assignment



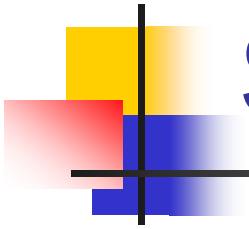
Compiler Code Generation

- Suggestion: isolate the actual compiler output (print) operations in a handful of routines
  - Modularity & saves some typing
  - Possibilities

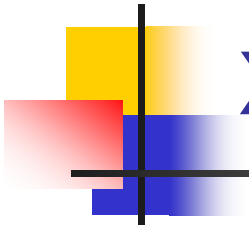
```
// write code string s to .asm output
   void gen(String s) { ... }
// write "op  src,dst" to .asm output
   void genbin(String op, String src, String dst) { ... }
// write label L to .asm output as "L:"
   void genLabel(String L) { ... }
```

  - A handful of these methods should do it

A Simple Code Generation Strategy



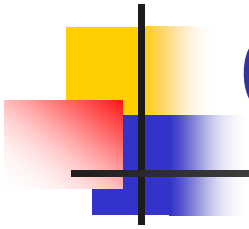
- Goal: quick ‘n dirty correct code, improve later if time
- Traverse AST primarily in execution order and emit code during the traversal
  - May need to control the traversal from inside the visitor methods, or have both bottom-up and top-down visitors
- Treat the x86 as a 1-register stack machine for now



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 winds up in eax
- If a value needs to be preserved while evaluating another expression, push eax, evaluate, then pop
  - Remember: always pop what you push
  - Will produce lots of redundant, but correct, code
- Examples below follow code shape examples, but with approximate gnu syntax – fix up as needed

Example: Generate Code for Constants and Identifiers



- Integer constants, say 17

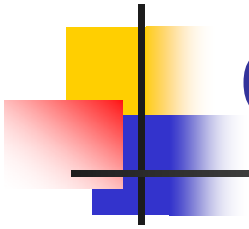
  gen(movl $17,%eax)

  - leaves value in eax

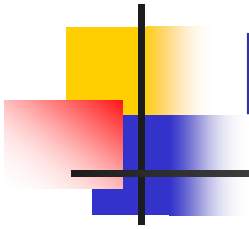
- Variables (whether int, boolean, or reference type)

  gen(movl var-offset(base-register),%eax)

Example: Generate Code for exp1 + exp1

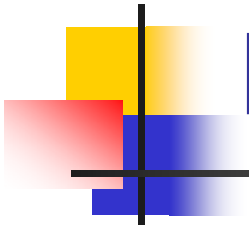


- Visit exp1
  - generates code to evaluate exp1 and put result in eax
- gen(pushl %eax)
  - generate a push instruction
- Visit exp2
  - generates code for exp2; result in eax
- gen(popl %edx)
  - pop left argument into edx; cleans up stack
- gen(addl %edx,%eax)
  - perform the addition; result in eax



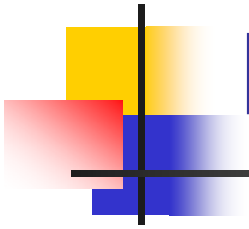
Example: var = exp; (1)

- Assuming that var is a local variable
  - visit node for exp
    - Generates code that leaves the result of evaluating exp in eax
  - gen(movl %eax,variable-offset(%ebp))



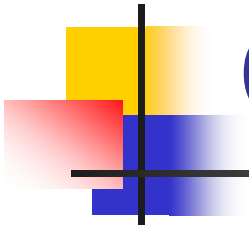
Example: var = exp; (2)

- If var is a more complex expression (object or array reference, for example)
  - visit var
  - gen(pushl %eax)
    - push reference to variable or object containing variable onto stack
  - visit exp
  - gen(popl %edx)
  - gen(movl %eax,appropriate_offset(%edx))



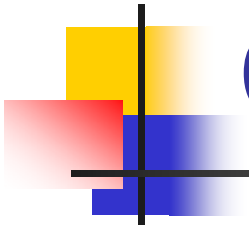
Example: return exp;

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



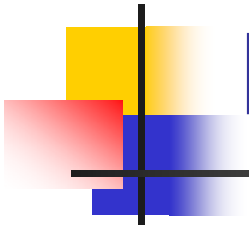
Control Flow: Unique Labels

- Needed: a String-valued method that returns a different label each time it is called (e.g., L1, L2, L3, ..., L42, ...)
  - 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, ...)



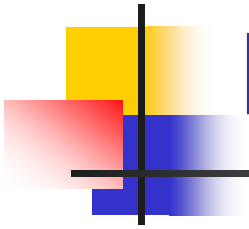
Control Flow: Tests

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



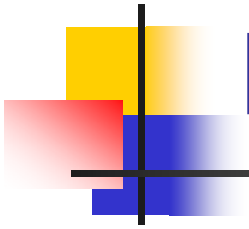
Example: while(exp) body

- 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"



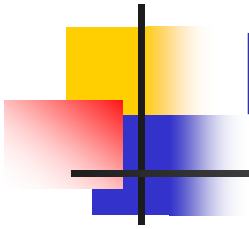
Example exp1 < exp2

- Similar to other binary operators
- Difference: context is a target label and whether to jump if true or false
- Code
  - visit exp1
  - gen(pushl %eax)
  - visit exp2
  - gen(popl %edx)
  - gen(cmp %edx,%eax)
  - gen(condjump targetLabel)
    - appropriate conditional jump depends 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



More to come...

- Still need to talk about methods, dynamic dispatch, object creation, etc.
  - See lecture slides
- Look at the codegen project writeup for suggestions about how to start small and build up to a complete compiler.