CS 412/413

Introduction to
Compilers and Translators
Andrew Myers
Cornell University

Lecture 38: Compilation strategies
3 May 00

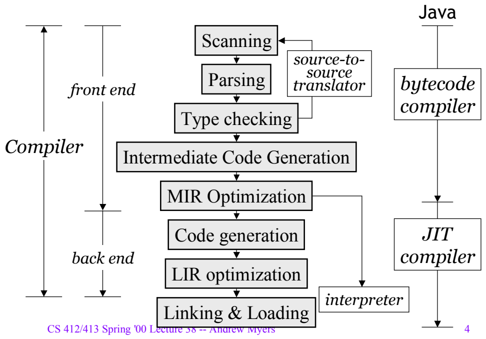
Administration

- Design reports due Friday
- Current demo schedule on web page
  - send mail with preferred times if you haven't signed up yet
  - keep on eye on the schedule!

Why build a compiler?

- You can design your own programming language
- *Domain-specific languages* can be designed for problems being solved
  - Code is shorter, easier to maintain: language has the right concepts baked in
  - Faster: can use optimize using special knowledge of language semantics
- This lecture: how to make it a little easier…

Compilers



Architectural independence

- Source-to-source translator: compile from source to another high-level language (e.g. C), let other compiler deal with code gen, etc.
- Compile from source to an intermediate code format for which a back end already exists (ucode, RTF, LCC, …)
- Compile from source to an executable intermediate code format, interpret:
  - abstract syntax tree
  - bytecodes (stack or register machine)
  - threaded code

Source-to-source translator

- Idea: choose well-supported high-level language (e.g. C, C++, Java) as target
- Translate AST to high-level language constructs instead of to IR, pass translated code off to underlying compiler
- Advantage: easy, can leverage good underlying compiler technology. Examples: C++ (to C), PolyJ (to Java), Toba (JVM to C)
- Disadvantages: target language won't support all features, optimization harder in target language, language may impose extra checks

Compiling to C

- C doesn't impose extra checks, is reasonably close to assembly, widely available (but can't support static exception tables
- *Mismatch*: no statements underneath expressions; must translate to canonical form in one step
- Translation of expression into C (or Java) is:
  - sequence of statements to be executed
  - expression to be evaluated afterward

$$\mathscr{E}\llbracket e \rrbracket = \{\, s_1; \ldots; s_n; \,\} \;\; ; \;\; e'$$
$$\mathscr{S}\llbracket s \rrbracket = \{\, s_1; \ldots; s_n; \,\}$$

Translation rules

- Translation still can be performed by recursive traversal of AST
- Some Iota → C rules:

assignment

$$\frac{\mathscr{E}\llbracket e \rrbracket = s \;\; ; \;\; e'}{\mathscr{E}\llbracket id = e \rrbracket = s \;\; ; \;\; id = e'}$$

block

$$\frac{\mathscr{E}\llbracket s_i \rrbracket = s_i' \;\; ; \;\; e_i' \quad ^{i \in 1..n}}{\mathscr{E}\llbracket (\, s_1; \ldots; s_n \,) \rrbracket = \{\, s_1'; \ldots; s_n'; \,\} \, ; \, e_n'}$$

Translating to Java

- Same problems as C, plus: Java is type-safe (good in a HLL, not so good in an intermediate language!)
- May need to use casting and `instanceof` expressions in generated code
  - dynamic type discrimination: slow

Back Ends

- Several standard intermediate code formats exist with back ends for various architectures—can reuse back ends
  - p-code: very old stack machine format
  - UCODE: old Stanford/MIPS stack machine format
  - Java bytecode: new stack machine format
  - RTL: GNU gcc, etc.
  - SUIF: Stanford format for optimization
  - LCC: Lightweight C compiler

Intermediate code formats

- Quadruples
  - compact, similar to machine code, good for standard optimization techniques
- Stack machine
  - E.g., Java bytecode format
  - easy to generate code for
  - hard to optimize directly
  - can be converted back into quadruples
  - used by some (sort of) high-level languages: FORTH, PostScript, HP calculators

Stack machine format

- Code is a sequence of stack operations (not necessarily the same stack as the call stack)

push CONST : add CONST to the top of stack

pop : discard top of stack

store : in memory location specified by top of stack store element just below.

load : replace top of stack with memory location it points to

+, *, /, -, ... : replace top two elems w/ result of operation

Generating code

- Stack operations mostly don't name operands (implicit): can code in 1 byte
- Expression is translated to code that leaves its value on the top of stack
- Translation of $E_1 + E_2$:

  $\llbracket E_1 + E_2 \rrbracket \;=\; \llbracket E_1 \rrbracket;\ \llbracket E_2 \rrbracket;\ +$
- Translation of $id = E$:

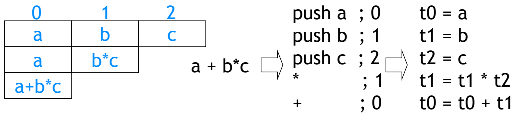
  $\llbracket\, id = E \,\rrbracket \;=\; \llbracket\, E \,\rrbracket;$ push $addr(id)$; store
- Bad code generation is easy

Compactness

- Values get "trapped" down low on stack especially with subexpression elim.
- Often need instruction that re-pushes element at known stack index on top
- Might as well have register operands!
- Result: not more compact than a register-based format; extra copies of data on stack too

Stack machine ⇒ quadruples

- At each point in code, keep track of stack depth (if possible)
- Assign temporaries according to depth
- Replace stack operands with quadruples using these temporaries



0	1	2
a	b	c
a	b*c	
a+b*c		

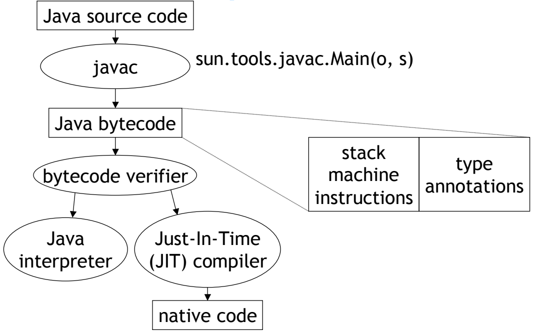
a + b*c ⇨

```
push a  ; 0
push b  ; 1
push c  ; 2
*       ; 1
+       ; 0
```

⇨

```
t0 = a
t1 = b
t2 = c
t1 = t1 * t2
t0 = t0 + t1
```

Java compilation model



Java source code

javac

sun.tools.javac.Main(o, s)

Java bytecode

stack machine instructions	type annotations

bytecode verifier

Java interpreter

Just-In-Time (JIT) compiler

native code

Verification

- Java security depends on
  - access only through public/protected methods
  - hidden private variables
  - unforgeable references to objects (capabilities)
- If Java program is not strongly typed, security of machine can be compromised!
- Java *bytecode verifier* checks Java bytecode to ensure strong typing: *typed intermediate language*
- Java Virtual Machine interpreter runs verified bytecode quickly, avoids run-time checks

JVM bytecode

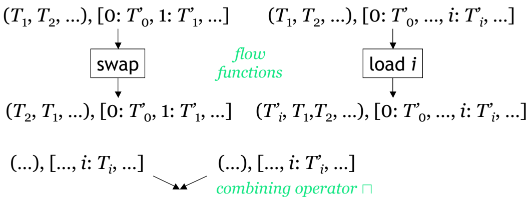
- stack-machine intermediate code
  - add, sub, mul, rem, div, ... : arithmetic
  - dup, swap, pop, ... : stack ops
- also has local registers/temporaries
  - load, store
  - untyped, reused for different types
- built-in object operations
  - invokevirtual, invokestatic, getfield, putfield, ...
  - types of methods, fields are declared
- control flow
  - ifeq, goto, ifne, ... : conditional branch
- How to show that code is type-safe? (efficiently!)

Type inference

- Type-checking bytecode: need to know
  - type of every stack entry
  - type of every local at every instruction
- Not present in bytecode file: inferred
- Start from
  - known argument, return types to method
  - object calls inside method
- Use forward data-flow analysis to propagate types to all bytecode instructions!
- Data-flow value is type of every stack entry, type of every local
- Meet is point-wise join in type hierarchy

Example

$$\text{Data-flow value} = \underbrace{(T_1, T_2, \ldots)}_{\textit{stack types}}, \underbrace{[0: T'_0, 1: T'_1, \ldots]}_{\textit{local types}}$$



$(T_1, T_2, \ldots), [0: T'_0, 1: T'_1, \ldots]$ $\xrightarrow{\text{swap}}$ $(T_2, T_1, \ldots), [0: T'_0, 1: T'_1, \ldots]$

$(T_1, T_2, \ldots), [0: T'_0, \ldots, i: T'_i, \ldots]$ $\xrightarrow{\text{load } i}$ $(T'_i, T_1, T_2, \ldots), [0: T'_0, \ldots, i: T'_i, \ldots]$

$(\ldots), [\ldots, i: T_i, \ldots]$ and $(\ldots), [\ldots, i: T'_i, \ldots]$ → $(\ldots), [\ldots, i: T_i \sqcup T'_i, \ldots]$

Object ⊔ int = ?

JIT compilers

- Particularly widely available back end(s) with well-defined intermediate code (JVM bytecode)
- Generate code by reconstructing registers from stack machine as discussed
- Inferred types allow better code
- Compilation is done on-the-fly: generating code quickly is essential → generated code quality is usually low
- HotSpot: new Sun JIT. High-quality optimization (*esp.* inlining and specialization), but used sparingly

Interpreters

- "Why generate machine code at all? Just run it. Processors are really fast"
- Options:
  - token interpreters (parsing on the fly) -- reallly slow (>1000x)
  - AST interpreters -- 300x
  - threaded interpreters -- 20-50x
  - bytecode interpreters -- 10-30x

AST interpreters

"Yet another recursive traversal"

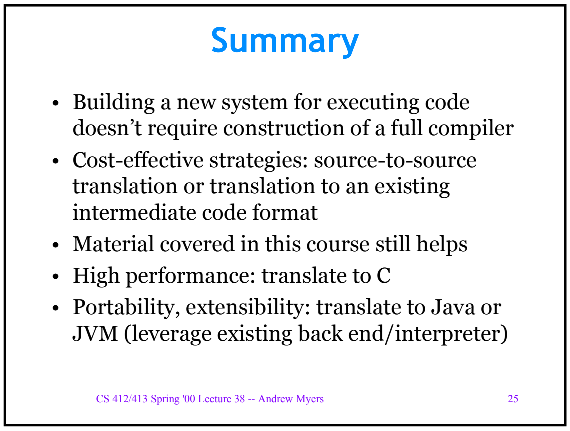
- For every node type in AST, add method `Object evaluate(RunTimeContext r)`
- Evaluate method is implemented recursively

```
Object PlusNode.evaluate(r)  {
    return left.evaluate(r).plus(right.evaluate(r)); }
```

- Variables, etc. looked up in r; some help from AST yields big speed-ups (*e.g.* pre-computed variable locations)
- Interpreter code broken into tiny methods w/ lots of method invocations: slow

Implementing bytecode interpreters

- Bytecode interpreter simulates a simple architecture (either stack or register machine)
- Interpreter state:
  - current code pointer
  - current simulated function return stack
  - current registers or stack & stack pointer
- Interpreter code is a big loop containing a switch over kinds of bytecode instructions
  - one big function: optimizer does good things
  - Avoid: recursion on function calls
- Result: 10-30x slowdown if done right



Summary

- Building a new system for executing code doesn't require construction of a full compiler
- Cost-effective strategies: source-to-source translation or translation to an existing intermediate code format
- Material covered in this course still helps
- High performance: translate to C
- Portability, extensibility: translate to Java or JVM (leverage existing back end/interpreter)