Please pick up
  - graded WA5, WA6
Topics

• Review WA5, WA6 problems
  – WA5 and WA6 sample solutions are on our web page

• Final checkpoint questions?
  – Submission instructions on the web

• Final Exam

• Bootstrapping a compiler

• Course wrap-up
Final Exam

• Administration
  – date and time
    • Tuesday May 2, 4 pm – 7 pm in two classrooms like the midterm
    • exam is similar in size to the midterm, but you have up to 3 hours to complete it
    • midterm and final have equal weight
  – rules (as before)
    • Open book, open notes, open computer
      – Computer: no search, no communication, no running your compiler

• Coverage
  – focus on second half of the class, but can include topics from first half
  – material from lecture notes and the programming project
    • mJAM, material on code generation for register machine
  – chapters in text, with emphasis on chapters 6 – 8
Compilers and Interpreters: diagrams

- your miniJava compiler
  - mJava \rightarrow mJAM
  - Java

- javac compiler on ia-32 machine
  - Java \rightarrow JVM
  - ia-32

- Compiling the miniJava compiler using javac on ia-32
  - mJava \rightarrow mJAM
  - Java \rightarrow JVM
  - ia-32
Compiling the mJAM interpreter

- mJAM interpreter

- java compiler on ia-32

- Compiling the TAM interpreter using java on ia-32
The miniJava compiler in action

- Compiling a miniJava program
  p1.mJava on ia-32

- Executing the compiled program on
  ia-32
The miniJava compiler is implemented in Java

Why?
- Java is a high-level language
  - can express key features of a compiler
    - advanced data structures
    - sophisticated modularization
    - design patterns
- Java is portable
  - can develop a miniJava compiler under Windows or Linux and run on either
Compiler implementation language

- What is the implementation language of Oracle’s Java compiler?

- The writers of the (Sun) Java compiler also preferred to implement their compiler in Java!

- A compiler for language L written in L is called a portable compiler
  - How do we compile such a compiler?
Compiler bootstrapping

- Given a pair of compilers from (high-level) language L into (machine) language M

  ![Diagram](image.png)

  portable compiler  native compiler

  we can conveniently construct
  - a compiler for L generating code for a different machine M’ (retarget)
  - a compiler for L’, an extension of language L (extend)
  - a better (e.g. more optimized) compiler for L into M (improve)

  using a technique called compiler bootstrapping
  - which yields a new pair of (portable, native) compilers
Retargeting a compiler

1. Write a new code generator to retarget the portable compiler to M’

2. Two-step bootstrap to construct the native compiler
Improving a compiler

1. Incorporate optimizations into the portable compiler

2. Two-step bootstrap to create (optimized) native compiler
1. Extend the compiler to handle new features of source language $L' \supseteq L$
   - but stick to features of $L$ in the *implementation*

2. Bootstrap (first half)
Extending the source language (2)

3. Rewrite the new compiler using the extended features of L’

4. Bootstrap (second half)
How to construct the first portable compiler?

- Incrementally, using repeated language extensions!

- Example: the first Pascal compiler
  - ETH Zurich, circa 1970
  - Machine: CDC 6000
  - Available languages: Scallop (CDC Assembler), Fortran
    - Initial attempt to write Pascal compiler using Fortran was a failure, and was discarded
  - A very simple and small compiler was written in a highly incomplete version of Pascal (P1)
    - It was translated by hand (!) to CDC Assembler
  - Thus started the bootstrap cycle!
The First Pascal Compiler

P1 = incomplete Pascal
P2 = full Pascal

hand translation

Two-step bootstrap

P2 → 6000

P1

language extension

P2 → 6000

P1

compiler improvement

P2 → 6000

P2

P1

6000

6000

6000

6000

6000

P2

6000

P2

6000

P2

6000

P2

6000

P2

6000

P2

6000

P1

6000

6000

6000

P1

6000

6000

6000
The Second Pascal Compiler

P2 = Original Pascal
RP = Revised Pascal

Another two-step bootstrap:

P2 = Original Pascal
RP = Revised Pascal
The Java compiler

• So is javac the *native* compiler for ia-32 (or x86-64) ?
  – Well, actually not
  – it’s actually more like a pre-jitted set of JVM classes for the *portable* compiler
  – but the compiler does use language improvement bootstrapping to make use of new language features

• JVM is a C++ program
  – it leverages the C++ compilers to generate a high quality interpreter on each architecture
  – and performs JIT compilation
  – it also provides native code for a boatload of basic library capabilities
    • e.g. GUIs, graphics
Language extension: example

• We have a compiler $C_1$ for a subset of C
  – handles escape sequences ‘\’ and ‘\n’ in character literals
  – produces MIPS assembly code

• We want to extend the subset of C
  – allow the escape sequence ‘\t’ (horizontal tab, ASCII code 0x09) to appear in character literals

• Relevant routine is `convert()`, in LexicalAnalysis section of $C_1$
  – converts character escape sequences in char literals to ascii codes
procedure convert()

```c
int convert() {
    int c = nextchar();
    if (c != '\\') return c;
    c = nextchar();
    if (c == '\') return '\';
    if (c == 'n') return '
';
    error();
}
```

Convert:
```
subu sp, sp, 24
sw ra, 16(sp)
jal nextchar ; $2 result
li $3, 0x5c ; backslash
bne $2, $3, $L1
jal nextchar ; $2 result
move $3, $2
li $2, 0x5c ; backslash
beq $3, $2, $L1
li $4, 0x0a ; ‘n’
beq $3, $4, $L1
jal error

$L1:
lw ra, 16(sp) ; return
addu sp, sp, 24 ; result in $2
jal ra
```

Code in $C_1$ compiler written in $C_1$ subset

$C_1$ compiled code for `convert()`
Extending C_1

• First try

```c
int convert() {
    int c = nextchar();
    if (c != '\\')
        return c;
    c = nextchar();
    if (c == '\\')
        return '\\';
    if (c == 'n')
        return '\n';
    if (c == 't')
        return '\t';
    error();
}
```

Generates a compile time error
Where?
Extending \( \text{C}_1 \)

- **Second try**

```c
int convert() {
    int c = nextchar();
    if (c != '\\')
        return c;
    c = nextchar();
    if (c == '\\')
        return '\\';
    if (c == 'n')
        return \n;
    if (c == 't')
        return 0x09;
    error();
}
```

- The \( \text{C}_1 \) compiler handles this just fine
  - \( \text{C}_1 \) compiles the extended compiler
  - produces a compiler \( \text{C}_2 \) that accepts \'t\' in char literals

```
convert:
    subu sp, sp, 24
    sw ra, 16(sp)
    jal nextchar ; $2 result
    li $3, 0x5c ; backslash
    bne $2, $3, $L1
    jal nextchar ; $2 result
    move $3, $2
    li $2, 0x5c ; backslash
    beq $3, $2, $L1
    li $4, 0x6e ; 'n'
    li $2, 0x0a ; '\n'
    beq $3, $4, $L1
    li $4, 0x74 ; 't'
    li $2, 0x09 ; 0x09
    beq $3, $4, $L1
    jal error
    lw ra, 16(sp) ; return
    addu sp, sp, 24 ; result in $2
    j ra
```

$\text{L1}$:
Completing the bootstrap

- $C_2$ will now be able to compile the preferred version of `convert()`
  - generates a third compiler $C_3$
  - now discard $C1$, $C2$ and retain the clean version of `convert()` in $C_3$

```c
int convert() {
    int c = nextchar();
    if (c != '\\')
        return c;
    c = nextchar();
    if (c == '\')
        return '\';
    if (c == 'n')
        return '
';
    if (c == 't')
        return '	';
    error();
}
```
So what happened?

• Some knowledge was “baked into” the (portable, native) compiler pair
  – The relationship between ‘\t’ and its character code 0x09 is no longer visible in the portable compiler
  – Yet the native compiler somehow reproduces it

• Is this OK?
  – It’s great for compiler development
  – It’s not ok for computer security!
    • The compiler can contain an embedded virus that propagates itself to future compilers
      – not visible in the portable compiler
      – but propagated into binaries
      – and into binaries of detectors!
    • “Reflections on trusting trust” – Ken Thompson 1984
Generating a compiler from an interpreter

• Partial evaluator
  – a kind of JIT compiler, but highly optimizing
  • Suppose we have \( \text{prog}(x,y) \), partial evaluator \( \text{PE} \) and known input \( x \)
  • \((\text{PE} \ \text{prog} \ x)\) is a \textit{residual program} such that
  • \((\text{PE} \ \text{prog} \ x)(y) = \text{prog}(x,y)\)
    – The partial evaluator “specializes” \( \text{prog} \) for known input \( x \)

• Bootstrapping a partial evaluator: the “Futamura projections”
  – First projection: \textit{compiles} a program \( P \) in language \( L \) given an interpreter for \( L \)
  • \((\text{PE}_{L} \ \text{interpreter}_{L} \ \text{P}_{L})(x) = \text{P}_{L}(x)\)
  – Second projection: builds a \textit{compiler} given an interpreter
    • \((\text{PE}_{L} \ \text{PE}_{L} \ \text{interpreter}_{L})(\text{P}_{L})(x) = (\text{PE}_{L} \ \text{interpreter}_{L} \ \text{P}_{L})(x)\)
  – Third projection: builds a \textit{compiler generator}
    • \((\text{PE}_{L} \ \text{PE}_{L} \ \text{PE}_{L})(\text{interpreter}_{L}) = (\text{PE}_{L} \ \text{PE}_{L} \ \text{Interpreter}_{L})\)
Back to something real …

• miniJava compiler
  – Built a few years ago by Bill Lewis
  – miniJava including multidimensional arrays and floating point arithmetic and lots of other features
    • also able to link to external graphics code
  – targeted to .NET (Microsoft’s virtual machine)

(1) Question about open source implementation of .NET (mono) ?
  • Answered by dusting off the miniJava compiler
  • and substituting mono for .NET
  • It runs!

(2) Portability of the mono environment ?
  • Recompiled the mono environment for a Raspberry Pi Zero ($5)
  • It runs!
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x</th>
<th>y</th>
<th>x</th>
<th>y</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.95</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>0.31</td>
<td>0.95</td>
<td>0.00</td>
<td>1.00</td>
<td>-0.81</td>
<td>0.81</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>-0.81</td>
<td>0.81</td>
<td>0.00</td>
<td>1.00</td>
<td>-0.81</td>
<td>-0.59</td>
<td>0.00</td>
<td>1.00</td>
</tr>
<tr>
<td>0.31</td>
<td>-0.95</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

setup(): perman
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

3.14
q = (true) ? 123 : 234; --> 123
class NativeCode: This is last static block

pi@raspberrypi:~/miniJava $ mono Compiler/miniJava.exe test1.mjava
pi@raspberrypi:~/miniJava $ ls -ltr test1.*
-rw-r--r-- 1 pi pi 62366 Apr 26 12:42 test1.mjava
-rw-r--r-- 1 pi pi 310937 Apr 27 10:34 test1.il
-rw-r--r-- 1 pi pi 23040 Apr 27 10:34 test1.exe
pi@raspberrypi:~/miniJava $
Wrap-up

• Compilers and interpreters
  – Critical components in modern programming
    • Integrated development environments, version control, configuration management, portable + secure programs
  – Their construction draws on all parts of CS
    • algorithms, data structures, automata theory, programming languages, graph theory, and software engineering

• (Much) we didn’t cover
  – error recovery
    • in the scanner and parser
  – optimization
    • loops and arrays, compiling for the memory hierarchy
  – complex programming language features
    • separate compilation (imports / exports / packages)
    • overloading and overriding
    • interfaces, generics, nested classes
    • concurrency
Course evaluation

• Please take the time to provide some feedback on this course
  – useful for me, to try to improve future offerings and teaching
  – useful for future students, to make decisions about the class

• Course evaluation mechanism
  – Online
    • closes tomorrow (Apr 28), so please do this now or soon
      … I won’t see any of these measures until well after grades are turned in

• Thanks!
  – Compilers are one of the many reasons why I love Computer Science!