Please pick up

- graded WA4 (back of the room)
  - by last name in three piles:
    - (A - H)
    - (J - R)
    - (S – Z)
Announcements

• **PA5 – final checkpoint**
  - submission directory will close Sun Apr 28 at midnight
    - supersedes deadline on the assignment
  - submission instructions are identical to PA4, plus
    • upload guide to your compiler (see assignment)
    • if you have implemented extra credit, include a Tests directory
  - readiness tester will be available shortly

• **Final exam**
  - Fri May 3, Noon – 3PM in SN014 (last name A-L), and FB009 (M-Z)
    • exam is similar in size to the midterm, but you have 3 hours to complete it
    • focus is on second half of the course, but will be integrative
    • midterm and final have equal weight
  - rules (same as before)
    • Open book, open notes
      - No computer, or communicating electronic devices
Topics today

• Exercise on register allocation in data flow graphs
  – will review solution in class today
  – exercise and sample solution are online

• Compilers, Interpreters, and Bootstrapping
  – material from Chapter 2 in our text
  – this material will not be in the exam
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

  Java \(\rightarrow\) JVM

  ia-32

  JVM

  ia-32
Compiling the mJAM interpreter

• mJAM interpreter

• javac compiler on ia-32

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

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

- Executing the compiled program on
  ia-32

```
  p1.mJava  mJava ➔ mJAM  p1.mJAM
        JVM
          JVM
              ia-32
                  ia-32
```

```
  p1.mJAM  mJAM
    JVM
      JVM
          ia-32
              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?

• Like us, the authors 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)

  - **portable compiler**
  - **native compiler**

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

  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
1. Incorporate optimizations into the portable compiler

2. Two-step bootstrap to create (optimized) native compiler
Extending the source language (1)

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)
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!

```
P1 6000
  hand-translation
P1 6000
```
The First Pascal Compiler

P1 = incomplete Pascal
P2 = full Pascal

P1 → 6000

language extension

P2 → 6000

compiler improvement

hand translation

Two-step bootstrap

P1 = incomplete Pascal
P2 = full 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. GUls, 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 '\n';
    error();
}
```

code in C₁ compiler
written in C₁ subset

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

C₁ 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 C₁

• Second try

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

• The C₁ compiler handles this just fine
  – C₁ compiles the extended compiler
  – produces a compiler C₂ 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, 0x74 ; 0x74
beq $3, $4, $L1
jal error
```

$L1$:
```
lw ra, 16(sp) ; return
addu sp, sp, 24 ; result in $2
j ra
```
Completing the bootstrap

- C₂ will now be able to compile the preferred version of convert()
  - generates a third compiler C₃
  - now discard C₁, C₂ and retain the clean version of convert() in C₃

```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();
}
```
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 prog(x,y), partial evaluator PE and known input x
    • (PE prog x) is a residual program such that
    • (PE prog x)(y) = prog(x,y)
      – The partial evaluator “specializes” prog for known input x

• Bootstrapping a partial evaluator: the “Futamura projections”
  – First projection: compiles a program P in language L given an interpreter for L
    • (PE_L interpreter_L P_L)(x) = P_L (x)
  – Second projection: builds a compiler given an interpreter
    • (PE_L PE_L interpreter_L)(P_L) (x) = (PE_L interpreter_L P_L) (x)
  – Third projection: builds a compiler generator 😊
    • (PE_L PE_L PE_L) (interpreter_L) = (PE_L PE_L Interpreter_L)
Back to something real …

- miniJava compiler
  - Built a few years ago by Bill Lewis
  - miniJava plus multidimensional arrays and floating point arithmetic and lots of other features
    - also able to link to an external libraries
  - 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!
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
    • Portability, IDEs, version control, configuration management, etc.
    • 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 through Connect Carolina
    • I won’t see any of these measures/comments until well after grades are turned in

• Thanks!
  – Compilers are one of the key tools that make Computer Science useful