Course Goals

- To gain understanding of the basic structure of programming languages:
  - Data types, control structures, naming conventions,...

- To study different language paradigms:
  - Functional (*Scheme*), imperative (*C*), logic (*Prolog*), parallel (*OpenMP*)
  - To ensure an appropriate language is chosen for a task

- To know the principles underlying all programming languages:
  - To make learning new programming languages easier
  - To enable *full* use of a programming language
  - To understand the implementation challenges of different programming constructs / features

Programming languages are **tools** ⇒ understand how to design or use them
Course Information

Prerequisites (summary):

- CS 205 (Introduction to Discrete Structures)
- CS 211 (Computer Architecture)

Important facts:

staff: Prof. Ulrich Kremer, TAs
lectures: Tue/Fri: sections 1,2,3: 10:20-10:40am, BBR-1071
        Tue/Fri: sections 5,6: noon-1:20pm, BBR-2071
recitations: attendance mandatory
        section 1, Tu 12:15-1:10pm, TIL-253
        section 2, Fr 12:15-1:10pm, TIL-105
        section 3, Th 3:35-4:30pm, HLL-009
        section 5, Tu 1:55-2:50pm, BRR-4071
        section 6, Fr 1:55-2:50pm, BRR-5105

Basis for grades (subject to changes):

10% homework / recitation
25% mid-term exam
35% final exam (cumulative)
30% three major programming projects
Course Information (Cont.)

- The textbook for this course is “Programming Language Pragmatics” by Michael L. Scott, 3rd Edition, Morgan Kaufmann (Elsevier), 2009.
- Additional (recommended) texts: see course web page

Course material is available on our class website at

www.cs.rutgers.edu/courses/314/classes/fall_2015_kremer

In addition, there is a news group (sakai web page) sakai.rutgers.edu. All questions regarding homeworks and projects MUST be posted here. You should read the sakai news group and look at the home page at least once every other day.

All programming will be done on the ilab cluster. Get yourself an ilab account (see link on bottom of our 314 website). Learn to do the normal things in Linux — edit, compile, ...
**Academic Integrity** (see our web page)
→ read-protect your directories and files (ilab)
→ no group projects
→ will use MOSS for detecting software plagiarism

14 weeks, no “make-up” work after the end of the course. If there is a problem, let me know immediately.

**IMPORTANT INFORMATION ⇒** will be posted on 314 web page and/or on sakai forums!

- Homworks and projects, and their grades
- Instructions of how to submit programming projects
- Partial credit for late project submissions
- Sample solutions for homeworks (sakai)

Email TAs or me:

- **Subject line** has to start with 314:
  e.g., 314: Question about my midterm exam

- **No** project and homework questions; post them on the sakai discussion forums;
Course Information (Cont.)

Special permission numbers

- Put your name (and email address!) on the list. List all possible sections for you. **BE AS FLEXIBLE AS YOU CAN!**
- We have many requests for SP numbers this semester.
- Will try to accommodate everyone, but it may not be possible due to capacity limits of lecture halls.
I use lecture notes

- I try to moderate my speed
- You need to say STOP!
- All lecture notes are on the Web (PDF)
  - draft will be available before class, e.g., lec1.pdf
  - final version will have a mod suffix, e.g., lec1mod.pdf
- You should still take some notes, since not everything we will talk about in class will be in the notes, for instance examples.

I’ll tell you where we are in the book

- I don’t lecture directly from the book
- You need to read the book
- Going to the recitations is mandatory
- I strongly recommend coming to the lectures
What is the Purpose of a Programming Language?

A programming language is . . .

a set of conventions for communicating an algorithm. Horowitz

Purposes:

• specifying algorithm and data structures
• communicating algorithms among people
• establishing correctness (allow reasoning)
• but also: provide foundation for different notions of computation
Why Use Anything Besides Machine Code?

This is a C program that uses two one-dimensional arrays \texttt{a} and \texttt{b} of size \texttt{SIZE}. The arrays are initialized, and then a sum reduction is performed. The size of the arrays and the result of the sum reduction is printed out.

eexample.c

```c
#include <stdio.h>
#define SIZE 100
int main() {
    int a[SIZE], b[SIZE];
    int i, sum;

    for (i=0; i<SIZE; i++) {
        a[i] = 1;
        b[i] = 2;
    }
    sum = 0;
    for (i=0; i<SIZE; i++)
        sum = sum + a[i] + b[i];

    printf("for two arrays of size %d, sum = %d\n", SIZE, sum);
}
```
Why Use Anything Besides Machine Code?

Compiler: gcc -O3 -S example.c ⇒ example.s

```assembly
.file "example.c"
.version "01.01"
gcc2_compiled:
.section .rodata.str1.32,"aMS",@progbits,1
.align 32
.LC0:  
.string "for two arrays of size %d, sum = %d\n"
.text
.align 4
.globl main
.type main,@function
main:
pushl %ebp
movl %esp, %ebp
xorl %eax, %eax
subl $808, %esp
movl $99, %edx
.align 2
.L21:
movl $1, -408(%ebp,%eax)
movl $2, -808(%ebp,%eax)
addl $4, %eax
decl %edx
jns .L21
xorl %ecx, %ecx
xorl %eax, %eax
movl $99, %edx
.align 2
.L26:
addl -408(%ebp,%eax), %ecx
addl -808(%ebp,%eax), %ecx
addl $4, %eax
decl %edx
jns .L26
pushl %eax
pushl %ecx
pushl $100
pushl $.LC0
call printf
addl $16, %esp
leave
ret
.Lfe1:  
.size main,.Lfe1-main
.ident "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3 2.96-112)"
```
Why Use Anything Besides Machine Code?

```
gcc -o example.o -O3 example.c; strip example.o;
objdump -d example.o
```

objdump: example.o: No symbols

```
example.o: file format elf32-sparc

Disassembly of section .text:

```
00000000 <.text>:
  00010444 <.text>:
10444: bc 10 20 00 clr %fp
10448: 00 03 a0 40 ld [ %sp + 0x40 ], %l0
1044c: a0 03 a0 44 add %sp, 0x44, %l1
10450: 9c 23 a0 20 sub %sp, 0x20, %sp
10454: 80 90 00 01 tst %g1
10458: 02 80 00 04 be 0x10468
1045c: 90 10 00 01 mov %g1, %o0
10460: 20 40 00 04 c call 0x20770
10464: 01 00 00 00 nsp
10468: 11 00 00 41 sethi $rhs(0x10400), %o0
1046c: 90 12 22 0d or %o0, 0x248, %o0 ! 0x106d8
10470: 40 00 00 40 c call 0x20770
10474: 01 00 00 00 nsp
10478: 80 00 00 91 call 0x106bc
1047c: 01 00 00 00 nsp
10480: 90 10 00 10 mov %l0, %o0
10484: 92 10 00 11 mov %l1, %o1
10488: 95 2c 20 02 sll %l0, 2, %o2
1048c: 94 02 a0 04 add %o2, 4, %o2
10490: 17 00 00 82 sethi %hi(0x20800), %o3
10494: 96 12 00 08 or %o3, 0x8, %o3 ! 0x106d8
10498: 40 00 00 4e call 0x105d8
1049c: 01 00 00 00 nsp
104a0: 80 00 00 20 call 0x105d8
104a4: 01 00 00 00 nsp
104a8: 40 00 00 45 c call 0x2077c
104ac: 01 00 00 00 nsp
104b0: 40 00 00 46 c call 0x20788
104b4: 01 00 00 00 nsp
104b8: 81 c3 c0 08 retl
104bc: ae 03 c0 17 add %o7, %l17, %l17
104c0: 9d e3 bf 90 save %sp, -112, %sp
104c4: 11 00 00 00 sethi %hi(0), %o0
104c8: 2f 00 00 40 sethi %hi(0x10000), %l7
104cc: 7f ff ff fb call 0x104b8
104d0: ae 05 e2 54 add %l7, 0x254, %l7 ! 0x10254
104d4: 90 12 20 0c or %o0, 0xc, %o0
104d8: d2 02 00 00 ld [ %o2 ], %o1
104dc: d0 24 00 00 st %o0, [ %l1 ]
104e0: 80 a2 a0 00 cmp %o2, 0
104e4: 12 80 00 23 bne 0x10570
104e8: 00 00 00 38 sethi %hi(0), %o0
104ec: 90 12 20 10 or %o0, 0x10, %o0 ! 0x10
104f0: d2 05 c0 08 ld [ %l17 + %o0 ], %o2
104f4: d2 02 00 00 ld [ %l2 ], %o1
104f8: a0 02 00 00 ld [ %o1 ], %o2
104fc: 80 a2 00 00 cmp %o2, 0
10500: 02 80 00 0f be 0x1053c
10504: 00 00 00 38 sethi %hi(0), %o0
10508: a0 10 00 0a mov %o2, %l0
1050c: a0 04 00 00 ld [ %l0 ], %o0
10510: 90 02 00 04 add %o0, 0x4, %o0
10514: 00 24 00 00 st %o0, [ %l10 ]
10518: d2 02 3f fc ld [ %o0 + -4 ], %o1
1051c: fe c2 40 00 call %o1
10520: 01 00 00 00 nsp
10524: 00 04 00 00 ld [ %l0 ], %o0
10528: d2 02 00 00 ld [ %o0 ], %o1
1052c: 80 a2 60 00 cmp %o1, 0
10530: 4f ff ff f9 bne 0x10514
10534: 90 02 00 04 add %o0, 0x4, %o0
10538: 11 00 00 00 sethi %hi(0), %o0
1053c: 90 12 20 0c or %o0, 0x10c, %o0 ! 0x10c
10540: d2 05 c0 08 ld [ %l17 + %o0 ], %o1
10544: 80 a2 60 00 cmp %o1, 0
10548: 02 80 00 05 be 0x1055c
```
CS 314  Fall’15 lecture 1, page 12
Why Use Anything Besides Machine Code?

Need for high-level programming languages for

- Readable, familiar notations
- Machine independence (portability)
- Consistency checks during implementation
- Dealing with scale

The art of programming is the art of organizing complexity. Example: *Dijkstra*, 1972

However:

- Acceptable loss of efficiency

First FORTRAN compiler built by IBM, in 1957, translated into code as efficient as hand-coded code. *John Backus*
Why Learn More than One Programming Language?

- Each language encourages thinking about a problem in a particular way.
- Each language provides (slightly) different expressiveness & efficiency.

⇒ The language should match the problem.
- Languages give insights into the foundations of computation

Why Learn About Programming Language PRINCIPLES?

A programming language is a tool.

Studying the design of a tool leads to:

- Better understanding of its functionality and limitations.
- Increased competence in using it.
- Basis for lots of other work in computer science.
Computational Paradigms

**Imperative:**
Sequence of state-changing actions.

- Manipulate an abstract machine with:
  1. Variables naming memory locations
  2. Arithmetic and logical operations
  3. Reference, evaluate, assign operations
  4. Explicit control flow statements
- Fits the von Neumann architecture closely
- Key operations: *Assignment* and “*Goto*”

**Functional:**
Composition of operations on data.

- No named memory locations
- Value binding through parameter passing
- Key operations: *Function application* and *Function abstraction*

Basis in *lambda calculus*
Computational Paradigms (Cont.)

Logic:

Formal logic specification of problem.

- Programs say *what* properties the solution must have, not *how* to find it
- Solutions through reasoning process.
- Key operation: *Unification*

Basis in **first order predicate logic**

Object-Oriented:

Communication between abstract objects.

- “Objects” collect both the data and the operations
- “Objects” provide *data abstraction*
- Can be either imperative or functional
- Key operation: *Message passing or Method invocation*
Computational Paradigms (Cont.)

Event-Driven:
Objects are associated with events

- events are asynchronous
- arrival of an event triggers action
- main applications: GUI, simulations
- Key operation: event handling

Parallel:
Computations and data accesses at the same time

- functional (task/threads) and data parallelism
- different granularities: instruction, loop, or task level
- synchronization: locks, message passing, ...
- Key notions: control and data dependencies
Compilers

Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- need format for object (or assembly) code

Big step up from assembler – higher level notations
Syntax and Semantics of Prog. Languages

**Syntax:**
Describes what a legal program looks like

**Semantics:**
Describes what a correct (legal) program means

A formal language is a (possibly infinite) set of sentences (finite sequences of symbols) over a finite alphabet $\Sigma$ of (terminal) symbols: $L \subseteq \Sigma^*$

Examples:

- $L = \{ \text{identifiers of length 2} \}$ with $\Sigma = \{a, b, c\}$
- $L = \{ \text{strings of only 1s or only 0s} \}$
- $L = \{ \text{strings starting with $\$\$ and ending with $\#$, and any combination of 0s and 1s inbetween} \}$
- $L = \{ \text{all syntactically correct Java programs} \}$

**Claim:** The larger the language, the harder it is to formally specify the language. In other words, it get’s harder for each $i$: $L_1 \subset L_2 \subset L_3 \ldots \subset L_i \subset \ldots$

True or false?
Syntax and Semantics: How does it work?

Syntactic representation of “values”

What do the following syntactic expressions have in common?

XI
1011
B
\( \lambda f.x. ( f ( f ( f ( f ( f ( f ( f ( f ( f ( f x))))))))))))\)

$ \# \# \# \# \# \# \# \# \# \# \# \# \# \# \# \# \#

3 + 20 - (2 \times 6)$
Syntax and Semantics: How does it work?

Syntactic representation of “values”

What do the following syntactic expressions have in common?

- XI
- 1011
- B
- λfx.(f(f(f(f(f(f(f(f(f(λfx.x))))))))))
- $\#$
- 3 + 20 − (2 × 6)

Answer: They are possible representations of the integer value “11” (written as a decimal number)

What is computation?

Possible answer: A (finite) sequence of syntactic manipulations of value representations ending in a “normal form” which is called the result. Normal forms cannot be manipulated any further.
Things to Do

Things to do for next lecture:

• read Scott: Chapter 1 (covers today’s lecture)
• read Scott: Chapters 2.1 and 2.2; ALSU: Chapters 3.1 - 3.4
• get an ilab account
• learn to read Sakai news group

Recitations will start next week.