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, CUDA*)
  – 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: Wednesday: 9:50 - 11:10am, AB-2160
          Friday: 2:50 - 4:10pm, AB-2160
recitations: attendance mandatory, starts next week
          section 1, Wednesday 12:15-1:10pm, LSH-B267
          section 2, Friday 10:35-11:30am, TIL-253
          section 3, Friday 12:15-1:10pm, BE-253

Basis for grades (subject to changes):

10% homework / recitation
25% mid-term exam
35% final exam (cumulative)
30% three major programming projects
• The textbook for this course is "Programming Language Pragmatics" by Michael L. Scott, 4th Edition, Morgan Kaufmann (Elsevier), 2015.

• Additional (recommended) texts: see course web page

Course material is available on our class website at

www.cs.rutgers.edu/courses/314/classes/fall_2017_kremer

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

Programming projects 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, …
Course Information (Cont.)

**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/resources)

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 piazza discussion forums;
Special permission numbers

NO PRE-REQUISITE OVERRIDE. SORRY!

To get an SP number to get into the class, or change a section:

- Put your name (and email address!) on the list. List all possible sections for you. **BE AS FLEXIBLE AS YOU CAN!**
Course Information (Cont.)

Miderm and final exams

• **Midterm**, 80 minutes, closed book, closed notes
• **Final**, common exam hour (3 hours)

**Conflict**: Examples: Another exam at the same time, or more than two exams in “consecutive” exams periods;
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.

\texttt{example.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

```
.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
.p2align 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
.p2align 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:

00010444 <.text>:
  10444: bc 10 20 00 clr %fp
  10448: a0 03 a0 40 ld [ %esp + 0x40 ], %l10
  1044c: a2 03 a0 44 add %esp, 0x44, %l11
  10450: 9c 23 a0 20 sub %esp, 0x20, %esp
  10454: 80 90 00 01 tst %zl
  10458: 02 80 00 04 be 0x10468
  1045c: 90 10 00 01 mov %zl, %zl0
  10460: 40 00 40 c4 call 0x20770
  10464: 01 00 00 00 nop
  10468: 11 00 00 41 sethi %hi(0x10400), %o0
  1046c: 90 12 22 48 or %o0, 0x248, %o0 ! 0x106d8
  10470: 40 00 40 c0 call 0x20770
  10474: 01 00 00 00 nop
  10478: 40 00 00 91 call 0x106bc
  1047c: 01 00 00 00 nop
  10480: 90 10 00 10 mov %zl0, %o0
  10484: 92 10 00 11 mov %zl1, %zl1
  10488: 95 2c 20 02 add %zl0, 2, %zl2
  1048c: 94 02 a0 44 add %zl2, 0x44, %zl2
  10490: 17 00 00 82 sethi %hi(0x20800), %o3
  10494: 96 12 a0 08 or %o3, 0x28, %o3 ! 0x208a8
  10498: d4 22 c0 00 st %o2, [ %o3 ]
  104a0: 40 00 00 4e call 0x105d8
  104a4: 01 00 00 00 nop
  104a8: 40 00 40 b5 call 0x2077c
  104ac: 01 00 00 00 nop
  104b0: 40 00 40 b6 call 0x20788
  104b4: 01 00 00 00 nop
  104b8: 81 c3 e0 08 retl
  104bc: ae 03 c0 17 add %hi7, %l17, %l17
  104c0: 9d e3 bf 90 save %sp, -112, %sp
  104c4: 11 00 00 00 sethi %hi0, %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 05 c0 08 ld [ %l7 + %o0 ], %o1
  104dc: d4 02 40 00 ld [ %o1 ], %o2
  104e0: 80 a2 a0 00 cmp %zo2, 0
  104e4: 12 80 00 23 bne 0x10570
  104e8: 11 00 00 00 sethi %hi0(0), %zo0
  104ec: 90 12 20 10 or %zo0, 0x10, %zo0 ! 0x11
  104f0: d4 05 c0 08 ld [ %l17 + %zo0 ], %zo2
  104f4: d2 02 80 00 ld [ %zo2 ], %zo1
  104f8: a0 02 40 00 ld [ %zo1 ], %zo0
  104fc: 80 a2 20 00 cmp %zo0, 0
  10500: 02 80 00 0f be on1053c
  10504: 11 00 00 00 sethi %hi0(0), %zo0
  10508: d2 02 3f fc ld [ %zo0 + -4 ], %zo1
  10510: 9f c2 40 00 call %zo1
  10514: 01 00 00 00 nop
  10518: a0 10 00 0a mov %zo2, %l10
  1051c: d0 04 00 00 ld [ %l10 ], %zo0
  10520: d2 02 80 00 ld [ %zo0 ], %zo1
  10524: 80 a2 60 00 cmp %zo1, 0
  10528: d2 02 40 00 ld [ %zo1 ], %zo0
  1052c: 80 a2 00 00 cmp %zo0, 0
  10530: d2 02 00 00 ld [ %zo0 ], %zo1
  10534: 02 80 00 0f be on10514
  10538: 11 00 00 00 sethi %hi0(0), %zo0
  1053c: 90 12 20 1c or %zo0, 0x10c, %zo0 ! 0x10c
  10540: d2 05 c0 08 ld [ %l17 + %zo0 ], %zo1
  10544: 80 a2 60 00 cmp %zo1, 0
  10548: 02 80 00 05 be on1055c
Disassembly of section .init:

000106bc <.init>:
106bc: 9d e3 bf a0 save %sp, -96, %sp
106c0: 7f ff ff b7 call 0x10584
106c4: 01 00 00 00 nop
106c8: 7f ff ff e6 call 0x10660
106cc: 01 00 00 00 nop
106d0: 81 c7 e0 08 ret
106d4: 81 e8 00 00 restore

Disassembly of section .fini:

000106d8 <.fini>:
106d8: 9d e3 bf a0 save %sp, -96, %sp
106dc: 7f ff ff 79 call 0x104c0
106e0: 01 00 00 00 nop
106e4: 81 c7 e0 08 ret
106e8: 81 e8 00 00 restore

Disassembly of section .plt:

00020770 <.plt>:
...
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 $i$: $L_1 \subset L_2 \subset L_3 \dots \subset L_i \subset \ldots$. 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
\( \lambda f.x. (f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(x))))))))))))))))))))))) \)

\$ | | | | | | | | #

3 + 20 − (2 × 6)
Syntax and Semantics: How does it work?

Syntactic representation of “values”

What do the following syntactic expressions have in common?

- XI
- 1011
- B
- $\lambda f.x.(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(x)))))))))))))))))))$
- $3 + 20 - (2 \times 6)$

**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.
Syntax and Semantics: How does it work?

Here is a “game” (rewrite system):

**input**: Sequence of characters starting with $\$ and ending with $\#$, and any combination of 0s and 1s inbetween.

**rules**: You may replace a character pattern $X$ at any position within the character sequence on the left-hand-side by the pattern $Y$ on the right-hand-side: $X \Rightarrow Y$:

rule 1  $1 \Rightarrow 1 \&$

rule 2  $0 \Rightarrow 0 \$

rule 3  $\& 1 \Rightarrow 1 \$

rule 4  $\& 0 \Rightarrow 0 \&$

rule 5  $\$ \# \Rightarrow \rightarrow A$

rule 6  $\& \# \Rightarrow \rightarrow B$

Replace patterns using the rules as often as you can, **one at a time**. When you cannot replace a pattern any more, stop.
Syntax and Semantics: How does it work?

example input:
$ 0 0 \#$
\[0 0 \#\] is rewritten as \[0 0 \#\] by rule 2
0 \[0 \#\] is rewritten as 0 0 \[\#\] by rule 2
0 0 \[\#\] is rewritten as 0 0 \[\rightarrow A\] by rule 6
no more rules can be applied (STOP)

More examples:

\$ 0 1 1 0 1 \$
\$ 1 0 1 0 0 \$
\$ 1 1 0 0 1 \$

Questions

- Can we get different “results” for the same input string?
- Does all this have a meaning (semantics), or are we just pushing symbols?
Syntax without Semantics?

Syntax without semantics is not useful!

Two problems on rewrite systems in the first homework.
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.