Topic Notes: Overview of Languages

We begin the course with an overview of languages, both historical and current. We will consider some of the essential features of languages, and categorize them.

Categories of Languages

We will classify most languages as one or more of *imperative*, *functional*, *logic*, *object-oriented*, *markup*, as well as either *compiled* or *interpreted*.

Throughout the semester, we will also see languages that have developed in response to hardware advances and new ideas in programming.

1940s – program hardware directly

1950s – simple applications (hardware focus)

1960s/1970s – structured programming

- costs shifted from hardware to software
- complexity and size of software grew dramatically

1970s/1980s – data-oriented program design

1980s – object-oriented program design

- data abstraction + inheritance + polymorphism

1990s/2000s – network/web applications

2010s – mobile applications, heavy multithreading


Imperative Languages

Imperative languages were developed first, and their design is heavily influenced by the *von Neumann architecture* at the heart of nearly all computers.

A program for a von Neumann architecture boils down to a set of instructions and a loop that executes those instructions:

1. Fetch an instruction

2. Update next instruction location

3. Decode the instruction

4. Execute the instruction

5. GOTO 1

Basic picture of the system:

scratchpad

Microsequencer (BRAIN!) **control store (microcode)**

ALU: arithmetic logic unit
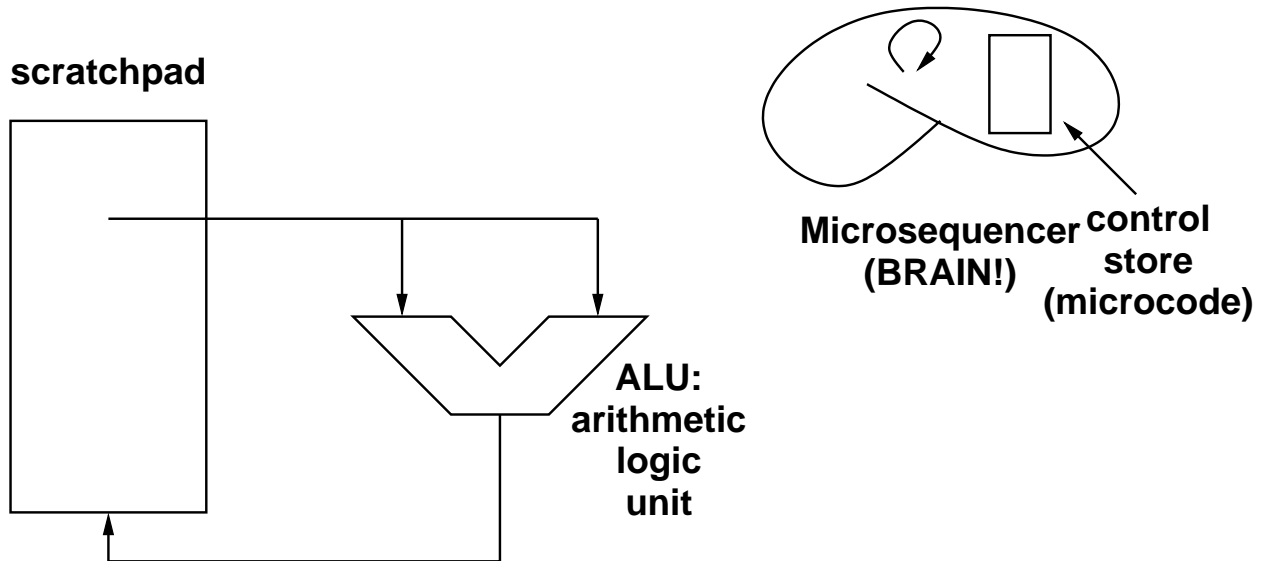
The ALU knows how to do some set of arithmetic and logical operations on values in the scratch-pad.

Usually the scratchpad is made up of a set of *registers*.

The micro-sequencer "brain" controls what the ALU reads from the scratchpad and where it might put results, and when.

This is what makes up the *central processing unit (CPU)*.

We expand this idea a bit to include memory and other devices:
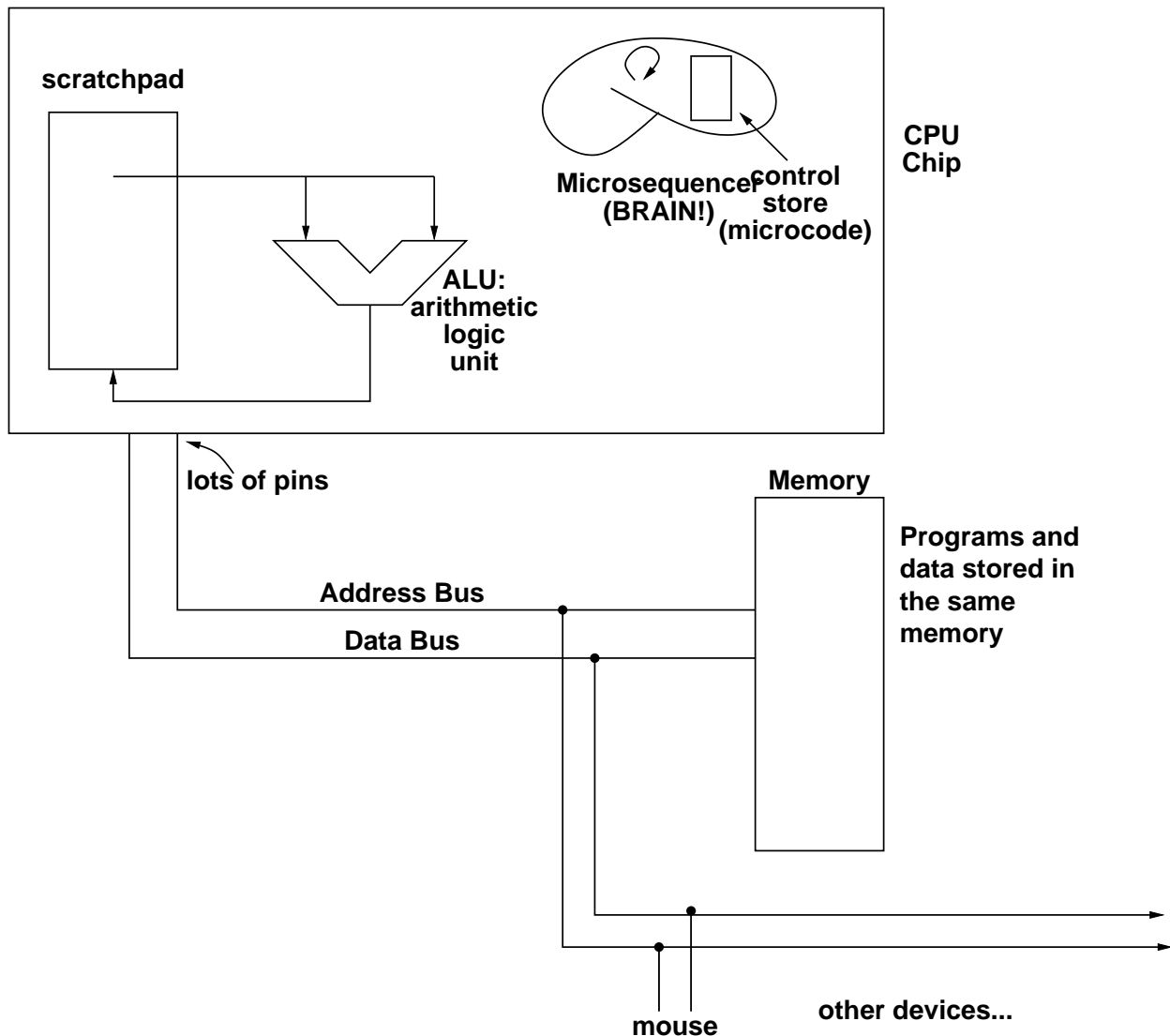
CPU interacts with memory and other devices on *buses*.

The details are the subject of a computer organization course. If you have had that course, you should be able to make connections between what we learn this semester and what you have seen there. If not, don't worry - you will be able to make those connections when you do.

But for us, it is important to understand that each instruction that can execute on our computer is capable of (some subset of) fetching information from registers and/or memory, applying some ALU operation to that information to obtain a result, and storing that result in a register and/or memory.

Both the programs and their data are stored in memory.

Quite naturally, early programming languages (and in fact, many current programming languages) have statements that perform functions that correspond to this architecture model.

- *Variables* mimic memory and registers – they store values

- *Assignment statements* allow us to modify variables

- *Arithmetic operations* compute values using the ALU

- *Iterative repetition* allows us to repeat sections of our program

- *Control structures* allow us to branch to different parts of the program conveniently

Examples of imperative languages include Fortran, C, C++, Pascal, C#, Java, Perl, and JavaScript.

Functional Languages

The primary mechanism for computing in a functional language is, unsurprisingly, the application of (often recursive) functions to parameters.

- pure functional programming has no variables or assignment statements!

- very convenient in some contexts

- not well-suited for others

- functional languages are usually interpreted rather than compiled

Examples include LISP and Scheme.

Logic languages

Logic languages use a completely different paradigm for programming. A program is specified as a set of rules, and it is up to the language to apply rules in an appropriate sequence to obtain a desired result.

- rules used to build a knowledge base

- perform queries against knowledge base

The most common example of a logic language is Prolog.

Object-oriented Languages

The object-oriented languages are not a disjoint category from the ones we have listed so far. In fact, object-oriented languages evolved from imperative languages.

Key features include:

- data abstraction

- inheritance

- polymorphism

- late binding

Examples of object-oriented languages include C++, Java, C#, Smalltalk, and Eiffel.

Markup and Web Languages

These are not really programming languages, but are worthy of a quick mention.

Web-based or application-specific markup languages specify layout of Web pages, database schemas, etc.

Common examples include HTML, XML, and XSLT.

Language Implementation

We will now take a brief look at how your program in a high-level language becomes the collection of bits in memory that can be executed by the architecture.

The simplest languages are *assembly languages*, which are not programming languages in the sense we are studying them this semester. An assembly language consists of a set of simple operations that can be performed that each correspond to a machine instruction that can be executed on a specific architecture.

For example, the following is an assembly language program for the MIPS architecture that populates a small array with powers of 2:

```
main:                                   # main() {
        la   $t0, ar                    #    get a pointer to ar into t0
        addi $t1, $0, 1                 #    value of 1 to place in first location
        sw   $t1, 0($t0)                #    place 1 into array[0]
        sll  $t1, $t1, 1                #    double value for next location
        sw   $t1, 4($t0)                #    place 2 into array[1]
        sll  $t1, $t1, 1                #    double value for next location
        sw   $t1, 8($t0)                #    place 4 into array[2]
        sll  $t1, $t1, 1                #    double value for next location
        sw   $t1, 12($t0)               #    place 8 into array[3]
        jr   $ra                        #    return control to the simulator
```

The details of the program are not important, but an assembly language program like this is used as input to an *assembler*, which converts each of the *assembly language instructions* to a single *machine instruction*.

In the case of MIPS, each will correspond to a unique 32-bit value, which when encountered by a MIPS processor, will cause the desired operation to occur.

Any program we can write in any language could be written in an assembly language, but we do not often do this.

High-level languages offer an easier, safer, and more portable way to write programs (using the ideas and constructs you already know and more we will study this semester).

Compiled Languages

It is important to understand how a program in a *compiled* high-level language works. We will look at an example in C, but similar procedures apply for C++, Fortran, and many other languages.

See Example:
`/home/cs433/examples/hello_c`

This C program defines one function, `main`, which calls one other function, `printf`.

For those unfamiliar, `main` is the function that starts executing when a C program starts, and `printf` is used to produce text output. It works much like Java's `System.out.println`, and is provided as part of C's *standard library*. The standard library is an extensive set of functions available for use by C programs. We will see more of these later.

For now, we want to think about what happens to turn this C source program into an executable program.

A C *compiler* translates the C source code into an intermediate code called *object code*. This object code is almost machine code but is missing some specifics. For example, in this case, the object code produced has the machine code for the `main` function, but has no mechanism to call that function. It also knows it needs to call a function `printf`, but does not have the code for that function.

For our example, we can perform just the translation from C source to object code with this command:

```
gcc -c hello.c
```

This produces a file called `hello.o`, which has the object code for the `main` function. The `-c` flag instructs the compiler to stop after creating the object file.

This file is not intended to be human readable, but if we look at it, we might recognize some artifacts from our source code, most notably the names of functions and any string constants we used.

We should also note that we are using a specific C compiler here, called GCC, for the GNU C Compiler, part of the GNU Compiler Collection. This is a free C (and many other language) compiler system that is available for many systems.

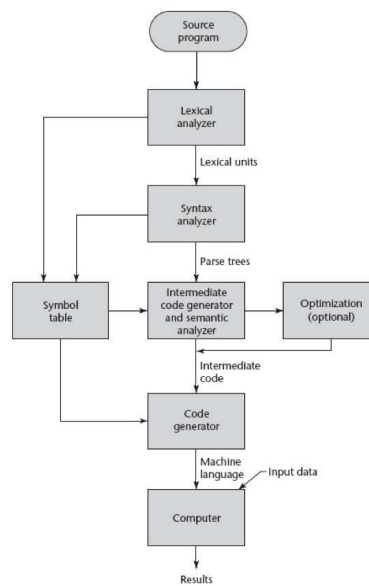We will look at more details of how a compiler works early this semester.

Figure 1.3 from Sebesta 2012.

To create an executable program, a second step called *linking* (done by a program called the *linker*) takes the object code from our `main` function, puts it together with precompiled object code for the `printf` function and the low-level code that knows how to start the `main` function, and produces an executable file.

For our example:

```
gcc -o hello hello.o
```

Invoked in this form (without the `-c` option from the previous step), `gcc` acts as a linker. The `-o` option allows us to specify the name of the executable file produced. We then list all of the object files (in this case, just the one) that contain object code that is needed by the program. `gcc` automatically brings in needed object code from the standard C library, in this case the `printf` function.

Once we have the executable, we can run it:

```
./hello
```

C programmers do not need to be concerned with all of these steps most of the time, but it is very helpful to be aware of them.

In fact, there are intermediate steps about which we should be aware. The first step is a *preprocessor* step that deals with compiler directives such as `#include` and `#define`. The compiler does not convert C source code directly to object code. It usually translates C code to the assembly language of the target architecture, then uses the assembler to convert to object code. We can ask our compiler to do just this step if we want to see the assembly code it generates.

If we invoke `gcc` with the `-E` option, it produces preprocessed C source code

```
gcc -E hello.c
```

If we invoke `gcc` with the `-S` option, it produces only the assembly language program that is normally sent to the assembler to produce object code.

```
gcc -S hello.c
```

After running this command, we can see the assembly language code for our computer, ready to be assembled and linked into an executable.

Interpreted Languages

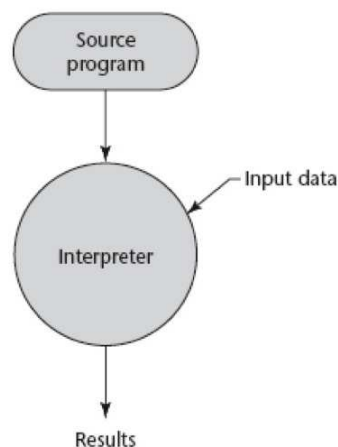At the other end of the spectrum is an *interpreted language*.

Figure 1.4 from Sebesta 2012.

Here, an *interpreter* (which is itself usually a program compiled as above) reads the source code for a program in an interpreted language and executes it.

We can try out some examples using:

On the web: Applesoft BASIC in JavaScript at
`http://www.calormen.com/applesoft/`

which uses JavaScript (another interpreted language – interpreted by a web browser!) as its interpreter.

While the lack of a compiler and linker eliminated the translation process, interpreted languages typically execute 10 to 100 times slower than compiled counterparts.

See Example:
`/home/cs433/examples/basic`

A Hybrid Approach: Compile for a VM

The situation is a bit different for Java. A Java compiler converts source code (your `.java` file) into an intermediate *byte code* (the corresponding `.class` file).

The byte code can then be interpreted on a Java virtual machine (JVM), rather than directly on the CPU as is done with C and similar languages. The JVM is the program that runs on the CPU.
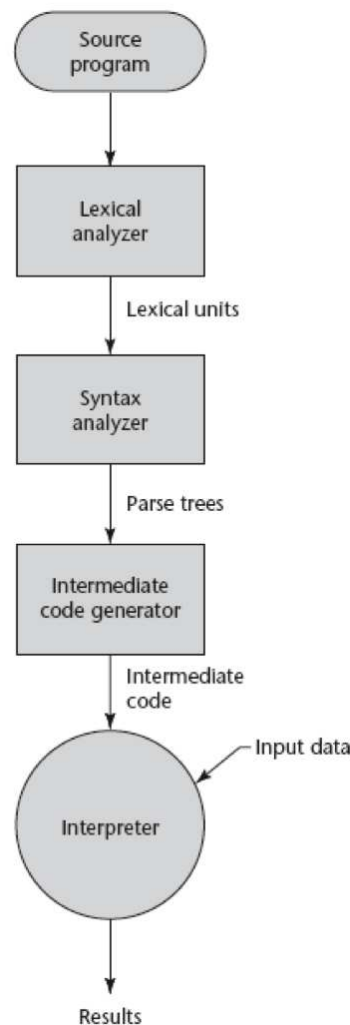

Figure 1.5 from Sebesta 2012.

There is no separate and explicit linking phase here. Instead, when the byte code for a Java class is executed, the JVM needs to be able to find any class files containing compiled byte code for any other classes it uses. This might be other classes you have compiled from your own Java source, or code from the Java libraries (like `Scanner`, `ArrayList`, etc.).

Note that the above is not completely true about modern Java implementations, which use "just in time" compilers and other technology to allow Java programs to run faster than they would if a JVM were to interpret byte code. We will likely say more about this later in the semester.

Layers of Abstraction

We can see the layered interface as provided by a typical modern computer system:
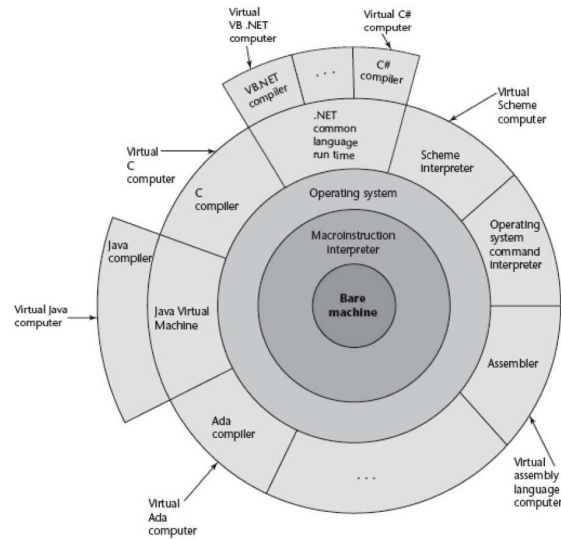

Figure 1.2 from Sebesta 2012.

Much of this diagram is the subject of an operating systems or computer organization course, but we can see how Java has an extra "layer" between the language and the hardware.

Historical Overview

Chapter 2 describes the historical development of programming languages. We will not cover this chapter in detail in class, but it is worth a read.

A diagram similar to the text's Figure 2.1:

On the web: Pixel's Programming Languages History Chart at
`http://rigaux.org/language-study/diagram.png`

Key ideas:

- Development of a compiled language: Fortran

- Development of a functional language: LISP

- Development of a more structured language: ALGOL

- Business records: COBOL

- String processing: SNOBOL

- Data abstraction: SIMULA 67

- A structures teaching language: Pascal

- Systems programming: C

- Logic programming: Prolog

- Department of Defense designs a language: Ada

- Development of object-oriented programming: Smalltalk, C++, Java

- Web-focused scripting languages: Perl, JavaScript, PHP, Python, Ruby

We will look back at many of the example languages from this chapter and their contributions as we cover other topics later in the semester.